

Microsoft

设计模式 ——.NET并行编程

(美) Colin Campbell Ralph Johnson 著
Ade Miller Stephen Toub

曹泽文 邹雪梅 李岸 译



清华大学出版社

Microsoft

设计模式 ——.NET并行编程

(美) Colin Campbell Ralph Johnson 著
Ade Miller Stephen Toub

曹泽文 邹雪梅 李岸 译



清华大学出版社
北京

内 容 简 介

本书结合大量的项目实践,介绍了与并行编程相关的概念、方法和应用。本书共7章:第1章主要介绍并行编程的基本概念与并行计算的基础理论,第2章主要介绍并行循环的知识,第3章介绍并行任务处理,第4章阐述并行合并计算的机理,第5章介绍future模式,第6章在前文的基础上深入探讨动态并行任务机制,第7章介绍并行编程的流水线机制。

本书适用于在.NET Framework上编写托管代码的程序员,包括在Visual C#、Visual Basic以及Visual F#上编写代码的程序员。本书不假定读者具有并行编程技术的预备知识。不过,读者需要熟悉C#的特征,如委托、lambda表达式、泛型以及语言集成查询(LINQ)表达式等。读者还至少应该对进程和线程的概念有基本的了解。

Parallel Programming with Microsoft .NET: Design Patterns for Decomposition and Coordination on Multicore Architectures by Colin Campbell, Ralph Johnson, Ade Miller and Stephen Toub (978-0-7356-5159-3)

Copyright © 2010 by Microsoft Corporation

Original English Language Edition Copyright © 2010 by Microsoft Corporation.

Published by arrangement with the original publisher, Microsoft Press, a division of Microsoft Corporation, Redmond, Washington, U.S.A.

本书中文简体版由Microsoft Press授权清华大学出版社出版发行,未经出版者书面许可,不得以任何方式复制或抄袭本书的任何部分。

北京市版权局著作权合同登记号 图字:01-2011-0438

本书封面贴有清华大学出版社防伪标签,无标签者不得销售。

版权所有,侵权必究。侵权举报电话:010-62782989 13701121933

图书在版编目(CIP)数据

设计模式——.NET并行编程/(美)坎贝尔(Campbell, C.)等著;曹泽文,邹雪梅,李岸译.——北京:清华大学出版社,2012

书名原文:Parallel Programming with Microsoft .NET: Design Patterns for Decomposition and Coordination on Multicore Architectures

ISBN 978-7-302-27997-6

I. ①设… II. ①坎… ②曹… ③邹… ④李… III. ①计算机网络—并行编译程序—程序设计 IV. ①TP393

中国版本图书馆CIP数据核字(2012)第018680号

责任编辑:文开琪 汤涌涛

封面设计:杨玉兰

责任校对:周剑云

责任印制:王静怡

出版发行:清华大学出版社

网 址: <http://www.tup.com.cn>, <http://www.wqbook.com>

地 址:北京清华大学学研大厦A座 邮 编:100084

社总机:010-62770175 邮 购:010-62786544

投稿与读者服务:010-62776969, c-service@tup.tsinghua.edu.cn

质 量 反 馈:010-62772015, zhiliang@tup.tsinghua.edu.cn

课 件 下 载: <http://www.tup.com.cn>, 010-62791865

印 装 者:北京嘉实印刷有限公司

经 销:全国新华书店

开 本:185mm×230mm

印 张:14.75

字 数:327千字

版 次:2012年4月第1版

印 次:2012年4月第1次印刷

印 数:1~4000

定 价:39.00元

产品编号:040287-01

译者序

随着计算机技术和计算方法的飞速发展,世界各国的高性能并行计算机的研制已取得长足进步。目前计算速度高达每秒百亿次、千亿次乃至数万亿次的高端并行计算机也已相继研制成功。正是由于并行计算的发展,许多以前无法求解和研究的问题,现在要解决它们已变得可能甚至轻而易举。因此,当我初读这本书时,就迫切希望能与别人分享建立在并行计算基础之上的并行编程技术,它也正是并行计算的精髓所在。

什么是“并行编程”呢?初次接触这一概念时我们可能会对这一描述感到惶恐甚至望而却步,但是仔细探索本书后,你会发现,它远没有我们想象的困难。所谓并行编程,简单地讲就是在并行计算机或分布式系统计算机等高性能系统上进行超级计算,以便合理调度资源。实际上,读者现在拥有的 PC 很可能就是一台双核甚至是四核的计算机,而其高效的计算处理能力正是源于本书将要阐述的并行编程技术。

本书内容丰富,几乎涵盖了并行编程的各个方面。一方面,本书既有对并行计算理论的基础原理及架构的阐述,也有对动态任务并行机制以及流水线技术的深入探讨,更重要的是本书每一章都有丰富的实例及示例代码,这有助于读者深入了解其原理与应用。另一方面,本书并非只是一本纯理论的书籍,因此真正善于学习的读者,应该适当地完成一些“课后练习”。对本书示例进行自我学习和探索,我相信您能够从本书获得宝贵经验。

翻译这本书使我受益良多,原作者在并行计算领域的丰富经验使得本书的探讨更加全面和细致,书中展示的许多实例都来源于真实项目的总结,实用性很强。作为译者,我十分高兴能有机会将作者的经验分享给大

家,希望它能帮助到更多的公司或者团队打造出更加高效而强大 的产品,同时也希望为计算机工作者提供一份可贵的参考。

参与本书翻译工作的有曹泽文、李岸、邹雪梅、邓振国、夏韵、徐连君、刘青、邓朝阳和邹剑波。在此感谢团队的每一个成员,没有你们的辛勤工作,这本书就不可能那么快地展现在大家的面前。另外,非常感谢陆昌辉老师的悉心指导。尽管团队的成员都尽心尽力,但由于经验所限,翻译工作尚有很多不足之处,恳请大家提出宝贵意见。

译 者



序

在并行计算出现的四十多年里，各国的并行计算专家将其应用到各领域中，如高能物理、工程应用及计算流体力学等。自早期的应用以来，我们在并行计算方面已经取得了巨大的进步。

并行计算领域的变化是由硬件快速发展驱动的，持续不断地提高处理器的时钟速度的时代已经结束了。取而代之的是摩尔定律所预言的，通过增加芯片密度来生产多核处理器或者具有多处理器内核的单芯片。目前，四核处理器非常普遍，而且依照这种趋势，在不久的将来，十核处理器也将面世。

在过去五年中，微软利用这项技术优势转化并且创造各种并行应用。其中包括为消息传递接口程序(MPI)而实现的 Windows 高性能集群(HPC)技术，为并行数据处理提供大聚簇数据处理(Map-Reduce)方式的 Dryad，按需提供计算内核的 Windows Azure 平台，为机器码提供的并行模式库(PPL)以及在.NET Framework 4 上加入的并行扩展。

多核计算影响着各层次和各方面的应用，从复杂的科学和设计问题到简单的消费类应用程序和新的人机接口。过去，我们常常开玩笑说“并行计算是未来的事，并且将永远都是”，但这种悲观主义现在被证明是错误的。并行计算从一项生僻的技术最终转变为应用程序开发者和整个 IT 产业的“明星”。

不过，这隐含着—个陷阱。为了加快应用程序的开发，程序员现在不得不将计算工作分成几块，以便充分利用多核处理器的性能，而这仍然需要专业技能。并行程序设计给绝大多数开发人员带来一个巨大的挑战，他们中的许多人还是第一次面对并行开发。因此，目前迫切需要通过实用的

方式获得这方面的知识，以便他们能将并行技术融入应用之中。

有两种可能的方案在从事计算机科学的同僚中非常受欢迎：设计一个新的并行编程语言，或者开发一个“健壮的”并行编译器。从学术上讲，这两种方法都相当有意思，但两者都不能成功地帮助非专业人员简化和普及并行工作。相比之下，另一种更实用的方法是为程序员提供一个库，隐藏并行程序设计的绝大部分复杂操作，然后再教程程序员如何使用它。

为此，Microsoft .NET Framework 并行扩展提供了一个比先前 API 更高级的编程模式。例如，程序员可以从任务的角度来思考，而不是从线程的角度，因而能够避开管理线程所带来的复杂性。Microsoft .NET 的并行程序设计通过让程序员置身于设计模式的背景，来告诉程序员怎样使用这些库。这样一来，应用程序开发者就能快速学习编写并行程序，从而即时获得性能优势。

本书强调了并行设计模式和最新的编程模型，我相信它将代表并行程序设计走向主流的重要的第一步。

Tony Hey

微软研究院，公司副总裁



前言

本书描述了并行程序设计中需要用到的模式，它们使用 Microsoft .NET Framework 4 中新加入的并行编程功能来进行并行设计，这种功能通常被称为并行扩展。你可以使用本书中的模式提高应用程序在多核计算机上的性能。在代码中使用这些模式能够使它们在现有机器上运行得更快，并且有助于适应将来的硬件环境——据预计，它们将融入越来越多的并行计算架构。

本书适合的读者

本书针对的读者对象是在 Windows 操作系统上利用 .NET Framework 编写托管代码的程序员。这包括在 Visual C#、Visual Basic 以及 Visual F# 上编写代码的程序员。本书不假定读者事先已了解并行编程技术。不过，读者需要熟悉 C# 的特征，如委托、lambda 表达式、泛型以及语言集成查询(LINQ)表达式等。读者还应该大致了解进程和线程的概念。

请注意：本书中的示例是用 C# 写的，并且使用了 .NET Framework 4 中的特性，包括任务并行库(TPL)和并行 LINQ(PLINQ)。不过，也可以将本书中呈现的概念应用到其他框架和库以及其他语言中。

完整的代码解决方案已经发布在 CodePlex 上，请参见 <http://parallelpatterns.codeplex.com/>。每个示例都有一个 C# 版本。除此之外，还有 Visual Basic 版本和 F# 版本。

为何在此时写作本书

Visual Studio 2010 开发系统实现了先进的并行编程功能,这使得并行程序设计的入门比以往任何时候都更加容易。

任务并行库(TPL)是为想要编写并行程序的.NET 程序员而开发的。它简化了在应用程序中应用并行和并发的过程。TPL 动态地调整并行度以最高效地利用所有可用的处理器。此外, TPL 还能协助您在.NET 线程池中划分和调度任务。该库提供了取消支持、状态管理以及其他服务。

并行 LINQ(PLINQ)是 LINQ to Object 的并行实现。PLINQ 实现了所有的 LINQ 标准查询操作符,将它们作为 System.Linq 命名空间的扩展方法,并额外加入了用于并行操作的操作符。PLINQ 是声明式的高层次接口,能够用于查询筛选、投影及聚合等操作。

Visual Studio 2010 包含了调试并行应用程序的工具。并行堆栈(Parallel Stack)窗口展示了应用程序中所有线程的堆栈调用信息。它能够让你自由地驰骋于各个线程与线程上的堆栈帧之间。并行堆栈窗口类似于线程窗口,只不过它显示的是每个任务的信息,而不是每个线程。通过 Visual Studio 分析器中的 Concurrency Visualizer 视图,可以看到应用程序如何与硬件、操作系统以及计算机上的其他进程交互。可以使用并发观测仪来定位性能瓶颈、处理器利用不足、线程争用、跨内核的线程迁移、同步延迟、I/O 冲突区域以及其他信息。

想要了解 Microsoft 可利用的并行技术的概况,请阅读附录 C。

使用代码的系统要求

本书使用的代码可以在 <http://parallelpatterns.codeplex.com/>找到。以下是对系统的要求:

- Microsoft Windows Vista SP1、Windows 7、Microsoft Windows Server 2008 或者 Windows XP SP3 (32 位或 64 位)操作系统
- Microsoft Visual Studio 2010 (使用 Concurrency Visualizer 需要 Ultimate 或 Premium 版本。可以通过 Concurrency Visualizer 来分析应用程序的性能), 其中包含了运行示例代码所需的 .NET Framework 4

如何使用本书

本书从具体的模式出发讲述了并行编程技术。图 0.1 展示了各个不同的模式以及它们之间的关系。其中, 数字代表该模式在本书哪一章中介绍。

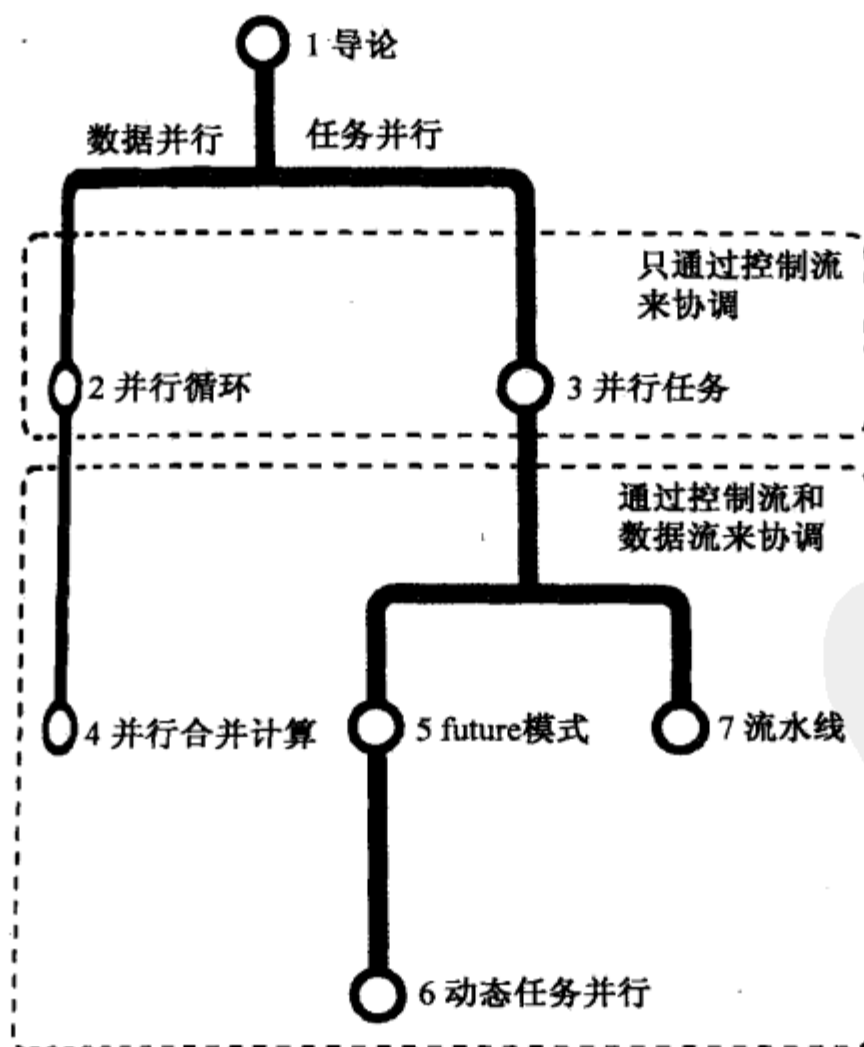


图 0.1 并行编程模式

在导论之后，本书从数据并行和任务并行两个方面讨论了并行编程技术。

并行循环和并行任务均运用程序的控制流作为协调和排列任务的方式。其他的模式则结合控制流和数据流两者来协调。控制流是指算法的步骤。数据流是指输入和输出的可用性。

导论

第 1 章阐述了意图使用并行技术来加快应用程序的运行速度的开发者们所面临的共同问题。本章解释了基本的概念，为后文做好铺垫。在 1.2.4 节中有一个表格可以帮助你选择适合你的应用的正确模式。

仅依赖控制的并行

第 2 章和第 3 章介绍了异步操作的排序仅受控制流限制的情况。

- 第 2 章介绍并行循环。想要在一个索引范围内或者一个集合的每个成员上执行相同的计算，并且成员之间没有依赖时，就使用并行循环。对于有依赖的循环，参见第 4 章。
- 第 3 章介绍并行任务。要执行几个不同的异步操作时，就使用并行任务。本章解释了为什么任务和线程适用于不同的目的。

依赖控制和数据的并行

第 4 章和第 5 章展示了既受控制流限制又受数据流限制的并发操作模式。

- 第 4 章介绍并行合并计算。并行合并计算模式适用于并行循环体中包含数据依赖的情况，例如计算总和或者在集合中查找最大值。
- 第 5 章介绍 `future` 模式。当操作产生的输出需要作为其他操作的

输入时，就要用到 `future` 模式。操作的顺序受限于数据依赖的有向图。有些操作是并行执行的，也有些是串行的，取决于输入的可用性。

动态任务并行和流水线

第 6 章和第 7 章讨论了一些更高级的应用场景。

- 第 6 章介绍动态任务并行。在某些情况下，操作是在计算过程中动态地添加到待办任务中的。这种模式可应用于某些领域中，包括图形算法和排序。
- 第 7 章介绍流水线。使用流水线将一个组件的连续输出提供给另一个组件的输入队列。当流水线被充满，或者超过一个组件同时处于活跃状态时，则将结果并行化。

支撑材料

除了对这些模式的描述，本书还提供了几个附录。

- 附录 A 针对一些常用面向对象模式(例如外观模式、装饰模式和数据仓库)在多核体系中的适应性给出了一些提示。
- 附录 B 简要介绍了如何在 Visual Studio 2010 中调试和分析并行应用。
- 附录 C 描述了并行编程方面的各种微软技术和框架。
- 术语表。术语表包含了本书中用到的术语。
- 参考文献。参考文献列出了文中提到的著作。

读者应当首先阅读第 1、2、3 章，大致了解基本的原理。尽管后面的章节是以逻辑顺序安排的，但从第 4 章起，每章都可以独立阅读。

本书中的标注是以特定的方式展示的，提示你应当特别小心的地方。

不要盲目应用本书介绍的模式。

如此处边栏所示。

试图运用一种新的工具或技术来解决面临的所有问题，而不管其适用性，这种想法是很诱人的。正如俗语所说，“当你有一把锤子的时候，任何东西看起来都像是图钉。”这种“什么都是图钉”的心态可能会导致非常不幸的结果。显然，大家都希望图 0.2 中的小兔子能够避免这种结果。

你当然也想在并行编程中避免不幸的结果。在应用中增加并行性需要花费时间，且增加了复杂性。为了取得好的效果，应当只在效益大于投入的部分应用并行化。

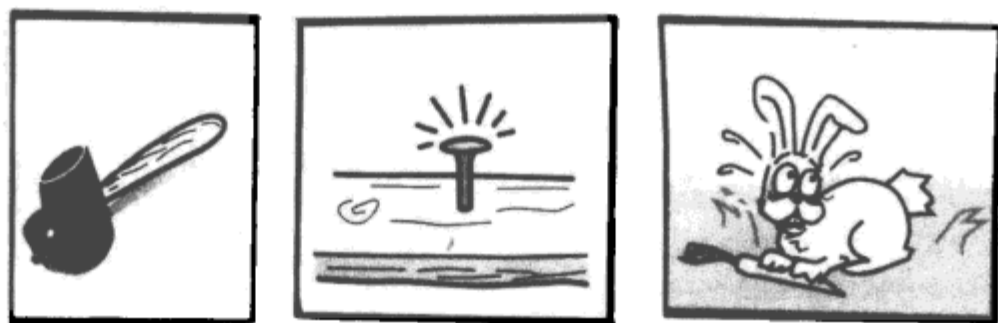


图 0.2 当你有一把锤子的时候，任何东西看起来都像是图钉

本书中没有包含的内容

比起 I/O 密集型的工作负载，本书更加关注的是计算密集型的负载。其目标是通过更好地利用计算机的可用内核，加快计算密集型的应用的运行速度。因此，本书没有花费太多的精力在 I/O 延迟上。不过，本书中也有关于负载均衡的讨论，且既包括计算密集型又包括 I/O 密集型(见第 7 章)。在第 5 章中还有一个关于用户界面的重要例子，该示例说明了带有 I/O 操作的任务的并发。

本书讨论了具有共享内存的单个多核节点中的并行性，而非集群中的并行，集群使用的是高性能计算(HPC)服务器的方法，在连通的节点上运用分布式存储。但是，想要在一个节点上利用并行优越性的集群程序员

也可以在本书中找到有用的例子，因为集群的每个节点可以有多个处理单元。

目标

读完本书，你应当能够完成以下任务。

- 回答每章后面的问题。
- 寻找书中是否有你的应用程序所适合的模式，如果有，要知道是否有望完成一个简单的并行实现。
- 理解你的应用什么时候适合使用其中的一个模式。在这个时候，你需要进行更多的阅读和研究，或者寻求专家的帮助。
- 如果你的并行实现不能工作，能够知道可能的原因，例如任务之间的依赖冲突或者错误的共享数据。
- 利用“扩展阅读”一节找到更多的资料。

资源
解
PDG

目 录

| | |
|-----------------------------------------|----|
| 第 1 章 导论 | 1 |
| 1.1 潜在并行性的重要性 | 2 |
| 1.2 分解、协调和可扩展共享 | 3 |
| 1.2.1 了解任务 | 4 |
| 1.2.2 协调任务 | 5 |
| 1.2.3 数据的可扩展共享 | 5 |
| 1.2.4 设计方法 | 6 |
| 1.3 选择恰当的模式 | 7 |
| 1.4 关于术语 | 8 |
| 1.5 并行性的极限 | 9 |
| 1.6 一些技巧 | 11 |
| 1.7 练习 | 12 |
| 1.8 扩展阅读 | 12 |
| 第 2 章 并行循环 | 15 |
| 2.1 基础知识 | 16 |
| 2.1.1 并行 for 循环 | 16 |
| 2.1.2 并行 ForEach 循环 | 17 |
| 2.1.3 并行 LINQ(PLINQ) | 18 |
| 2.1.4 预期 | 19 |
| 2.2 示例 | 21 |
| 2.2.1 信贷审查的顺序版本示例 | 22 |
| 2.2.2 使用 Parallel.ForEach 的信贷审查示例 | 23 |

| | | |
|-------|----------------------------|----|
| 2.2.3 | PLINQ 信贷审查示例 | 23 |
| 2.2.4 | 性能比较 | 24 |
| 2.3 | 变化形式 | 24 |
| 2.3.1 | 尽早中断循环 | 25 |
| 2.3.2 | 外部循环取消 | 29 |
| 2.3.3 | 异常处理 | 30 |
| 2.3.4 | 小循环体的特殊处理 | 31 |
| 2.3.5 | 控制并行度 | 33 |
| 2.3.6 | 在循环体中使用局部任务状态 | 35 |
| 2.3.7 | 对并行循环使用自定义的任务调度程序 | 36 |
| 2.4 | 反模式 | 37 |
| 2.4.1 | 步长不为 1 | 37 |
| 2.4.2 | 隐藏的循环体依赖 | 38 |
| 2.4.3 | 少量迭代的小循环体 | 38 |
| 2.4.4 | 处理器的超额申请和申请不足 | 38 |
| 2.4.5 | 混合 Parallel 类和 PLINQ | 39 |
| 2.4.6 | 输入枚举中的重复 | 39 |
| 2.5 | 设计说明 | 40 |
| 2.5.1 | 自适应分区 | 40 |
| 2.5.2 | 自适应并发 | 40 |
| 2.5.3 | 支持嵌套循环和服务端应用程序 | 41 |
| 2.6 | 相关模式 | 41 |
| 2.7 | 练习 | 42 |
| 2.8 | 扩展阅读 | 43 |
| 第 3 章 | 并行任务 | 45 |
| 3.1 | 基础知识 | 46 |
| 3.2 | 示例 | 47 |

| | |
|------------------------------|----|
| 3.3 变化形式 | 50 |
| 3.3.1 取消任务 | 50 |
| 3.3.2 处理异常 | 52 |
| 3.3.3 等待第一个任务完成 | 56 |
| 3.3.4 推测执行 | 57 |
| 3.3.5 使用自定义的调度方式创建任务 | 59 |
| 3.4 反模式 | 60 |
| 3.4.1 闭包捕获的变量 | 60 |
| 3.4.2 清理任务所需要的资源 | 61 |
| 3.4.3 避免撤销线程 | 62 |
| 3.5 设计说明 | 62 |
| 3.5.1 任务和线程 | 62 |
| 3.5.2 任务生命周期 | 62 |
| 3.5.3 编写自定义的任务调度程序 | 64 |
| 3.5.4 未观测到的任务异常 | 64 |
| 3.5.5 数据并行性和任务并行性之间的关系 | 65 |
| 3.6 默认任务调度程序 | 66 |
| 3.6.1 线程池 | 66 |
| 3.6.2 分散管理的调度技术 | 68 |
| 3.6.3 work stealing 策略 | 69 |
| 3.6.4 全局队列中的顶层任务 | 70 |
| 3.6.5 局部队列中的子任务 | 70 |
| 3.6.6 子任务的内联执行 | 71 |
| 3.6.7 线程注入 | 72 |
| 3.6.8 绕过线程池 | 74 |
| 3.7 练习 | 74 |
| 3.8 扩展阅读 | 75 |

| | |
|---------------------------------|-----|
| 第 4 章 并行合并计算..... | 77 |
| 4.1 基础知识 | 77 |
| 4.2 示例 | 80 |
| 4.3 变化形式 | 84 |
| 4.3.1 使用并行循环进行合并计算..... | 84 |
| 4.3.2 使用范围分割器进行合并计算..... | 87 |
| 4.3.3 使用带有范围选择的 PLINQ 合并计算..... | 89 |
| 4.4 设计说明 | 91 |
| 4.5 相关的模式 | 94 |
| 4.6 练习 | 94 |
| 4.7 扩展阅读 | 95 |
| 第 5 章 future 模式 | 97 |
| 5.1 基础知识 | 98 |
| 5.1.1 future | 98 |
| 5.1.2 延续任务 | 101 |
| 5.2 示例: Adatum 金融仪表板 | 102 |
| 5.2.1 业务对象 | 104 |
| 5.2.2 分析引擎 | 104 |
| 5.2.3 视图和视图模型..... | 110 |
| 5.3 变化形式 | 110 |
| 5.3.1 取消 future 和延续任务..... | 111 |
| 5.3.2 拥有多个先行任务的情况 | 111 |
| 5.3.3 使用.NET 异步调用和 future | 111 |
| 5.3.4 消除瓶颈 | 112 |
| 5.3.5 运行时修改图 | 112 |
| 5.4 设计说明 | 113 |
| 5.4.1 分解成 future 和延续任务..... | 113 |

| | | |
|-------|----------------------------|-----|
| 5.4.2 | 函数式风格 | 113 |
| 5.5 | 相关的模式 | 114 |
| 5.5.1 | 流水线模式 | 114 |
| 5.5.2 | 主/从(master/worker)模式 | 114 |
| 5.5.3 | 动态任务并行模式 | 114 |
| 5.5.4 | 离散事件模式 | 115 |
| 5.6 | 练习 | 115 |
| 5.7 | 扩展阅读 | 115 |
| 第 6 章 | 动态任务并行 | 119 |
| 6.1 | 基础 | 119 |
| 6.2 | 示例 | 121 |
| 6.3 | 变化形式 | 123 |
| 6.3.1 | while-not-empty 并行 | 123 |
| 6.3.2 | 任务链与父子任务 | 125 |
| 6.4 | 设计说明 | 126 |
| 6.5 | 练习 | 127 |
| 6.6 | 扩展阅读 | 127 |
| 第 7 章 | 流水线 | 129 |
| 7.1 | 基础 | 129 |
| 7.2 | 示例 | 134 |
| 7.2.1 | 顺序图像处理 | 134 |
| 7.2.2 | 图像流水线 | 135 |
| 7.2.3 | 运行特性 | 137 |
| 7.3 | 变化形式 | 139 |
| 7.3.1 | 取消流水线 | 139 |
| 7.3.2 | 处理流水线异常 | 141 |

- 7.3.3 利用多个生产者实现负载均衡.....144
 - 7.3.4 流水线和流.....146
 - 7.3.5 异步流水线.....147
- 7.4 反模式.....147
 - 7.4.1 线程饥饿.....147
 - 7.4.2 阻塞集合无穷等待.....147
 - 7.4.3 忘记 GetConsumingEnumerable()方法.....148
 - 7.4.4 采用其他生产者/消费者集合.....148
- 7.5 设计说明.....148
- 7.6 相关模式.....149
- 7.7 练习.....149
- 7.8 扩展阅读.....150

附录 A 改写面向对象模式.....151

附录 B 调试和分析并行应用程序.....183

附录 C 技术概览.....197

术语表.....201

参考文献.....211



第 1 章 导 论

CPU 计量器能够说明问题。例如，一个内核的使用率为 100%，而其他内核都是空闲的。或者应用程序是计算密集型的(即要占用大量 CPU 资源)，但你只使用了多核系统的一部分计算能力。如何解决这种问题呢？

简而言之，答案就是并行编程。像所有程序员一样，你可能熟谙编写顺序代码之道，但你会发现现在它不再满足你的性能要求了。要想有效地使用系统的 CPU 资源，必须把应用程序分割成块，使各块在同一时间运行。

并行编程在同一时间运行多个内核，以提高应用程序的速度。

这说起来容易做起来难。并行编程被誉为专家领域和敏感地带，很难重现软件缺陷。每个程序员似乎都有一些关于并行编程的趣事，可能由于某个神秘的错误，程序并没有像预期那样运行。

在编写并行程序时，这些故事应该能帮助你正视面临的问题和困难。幸运的是，已经有人提供了一些帮助。.NET Framework 4 引入了一种新的编程模型，大大简化了并行工作。后台是复杂算法的支持库，在多核架构中动态分配计算。此外，Visual Studio 2010 还包含调试和分析工具，以支持新的并行编程模型。

编写并行程序出了名的困难，但现在有了助力。

另一个帮助来源是经过验证的设计模式。本书借助任务并行库(TPL)和并行语言集成查询(PLINQ)，介绍了最重要而且使用最频繁的并行编程模型，并给出了这些模型的可执行代码示例。关于要从哪里入手，最好的起点就是阅读本书上的模式。看看问题的属性是否与后续各章中提出的六种模式相匹配。如果匹配，就更深入地探究相关模式，研究示例代码。

大多数并行程序都符合这些模式，你很可能能够成功地找到与你的问题相符的模式。如果不能使用这些模式，你可能遇到了一种更困难的情形，那么你将需要聘请专家或者查阅学术文献。

本书的示例代码可以在 <http://parallelpatterns.codeplex.com> 网站上找到。

1.1 潜在并行性的重要性

声明程序的潜在并行性，使执行环境可以在所有可用的内核中运行程序，不管是单核还是多核。

本书中的模式是表达潜在并行性的方法。这意味着在并行硬件可用时，程序会运行得更快，而并行硬件不可用时，程序运行的速度也几乎能与顺序程序相当。如果正确地组织了代码，运行环境可以自动地适应特定计算机上的工作负荷。这就是为什么本书的模式只表达潜在并行性。它们不保证在任何情况下都能并行执行。表示潜在并行性是.NET 编程模型背后的一个中心组织原则。我们要对它做一些解释。

我们可以为特定硬件编写并行应用程序。例如，游戏机平台的程序创作者具有关于硬件资源的详细知识，在运行期这些知识就派得上用场。他们预先知道内核的数量和内存架构的细节。利用平台提供的并行性的确切程度，编写游戏。充分考虑到硬件环境也是一些嵌入式应用程序的特征，如工业控制。这些程序的生命周期与设计来运行的特定硬件的生命周期相同。

不要执着地按应用程序的并行程度编码。你并不是总能预测到运行时会有多少内核可用。

与此相反，编写运行在通用计算平台上的程序时，如桌面工作站和服务端，这些硬件的特征几乎是不可预知的。你不一定知道会有多少个可用内核。你可能也无法预知会有哪些其他软件与你的应用程序同时运行。

即使最初知道应用程序的环境，它也会随着时间而改变。在过去，程序员猜想他们的应用程序在升级后的硬件中会自动运行得更快。你可以相

信这个假设，因为处理器时钟的速度在不断地提高。随着多核处理器的出现，处理器的时钟速度不再像过去那样随着新的硬件的出现一起提高。处理器设计的趋势是向着更多内核的方向发展。如果希望应用程序受益于多核世界硬件的进步，需要调整编程模型。你应该预料到今天的程序在几年后就会运行在多了很多内核的电脑上。强调潜在的并行性有助于程序“面向未来”。

未来的硬件趋势是更多的内核，而不是更快的时钟速度。

最后，你为这些可能性所做的计划不能对无法使用最新硬件的用户不利。你要让并行应用程序在单核电脑上运行的速度与用顺序代码编写的应用程序一样快。换句话说，从单核到多核，应用程序需要能够适应不同性能的硬件，能迎合现在和未来，这就是潜在并行性的动机。

一个精心编写的并行程序在只有一个可用内核的情况下，运行的速度几乎与顺序程序一样。

第 2 章描述的并行循环模式，是潜在并行性的一个例子。假如你有一个 for 循环，独立重复执行一百万次，那么把它们重复分配到可用内核上并行工作是有意义的。我们很容易发现，划分工作取决于内核的数量。通常情况下，循环的速度几乎与内核的数量成正比。

1.2 分解、协调和可扩展共享

本书中提到的模式包含一些共同的主题。你会看到，设计和实现一个并行应用程序的过程包括三个方面：把工作分解为离散的单元(即任务)的方法；由于这些任务并行运行，所以还需要协调这些任务的方法；执行任务时用于共享数据的可扩展性技术。

本书中描述的模式是设计模式。可以把这个设计模式应用在设计 and 实现算法以及构思应用程序的整体架构中。虽然示例应用程序规模小，但是它们所展示的原理同样适用于大型应用程序的架构。

1.2.1 了解任务

任务是工作的顺序单元。任务应该大，独立，数量足够多以使所有的内核都忙碌起来。

请记住，任务并不是线程。任务和线程采用不同的方法进行调度。与线程相比，任务与潜在并行性的概念更相一致。一个新线程立即就能为你的应用程序引入额外的并发性，而一个新任务仅仅带来额外的并发可能性。只有存在足够多的可用内核时，任务潜在的额外并发性才可能实现。

任务是协同工作的一系列顺序操作，它们共同完成一个更大的操作。谈论到如何构建并行程序时，有一点很重要，即确定任务的粒度，这有助于硬件资源的有效利用。如果选择的粒度太细，任务管理开销就占主要地位。如果粒度太粗，就可能会失去并行的机会，因为原本可以使用的内核却被闲置着。一般情况下，任务应该尽可能大，但它们之间应该相互独立，并且应该要有足够多的任务使内核都忙碌起来。在调试任务时，可能还需要用到试探法。要满足所有这些目标，有时需要折中设计。要把问题分解为任务，要求对算法和应用程序的结构方面有很好的了解。

这些准则的一个例子是并行光线追踪应用程序。光线追踪器通过模拟场景中的每一条光线的路径来构造一张合成图像。在并行程度上，单独的光线模拟有较好的粒度水平。把任务分解成较小的单元，例如，把光线模拟本身分解成独立的任务只会增加开销，因为光线模拟的数目已经足以占用所有的内核。如果任务的大小差异很大，通常需要任务数超过内核数，用这些任务来填补间隙。

把工作划分成少而大的任务的另一个优势是，这些任务之间的独立性通常优于划分为多而小的任务的独立性。相较于较小的任务，较大的任务之间共享局部变量或字段的可能性较小。遗憾的是，在依赖于大型可变对象图的应用程序中，如有许多公共类、方法和属性的大型对象模型中，结论可能相反。在这种情况下，较大的任务更容易出现更多共享数据的意外或其他副作用。

我们的总目标是在提供足以占用所有可用内核的任务的同时，把问题分解为不共享数据的独立任务。另外，在考虑内核数目时，应该想到未来的硬件将会有更多的内核。

1.2.2 协调任务

同一时间有多个任务在运行，这是非常有可能的。相互独立的任务可以并行运行，而有些任务只能在其他任务完成之后才可以开始。执行的顺序和并行程度受制于应用程序的底层算法。这种限制是由控制流(该算法的步骤)或数据流(输入输出的情况)引发的。

可能存在各种负责协调任务的机制。任务协调的方式依赖于使用的并行模式。例如，第 7 章所描述的管道模式的特点是使用并发队列协调任务。无论为协调任务选择什么机制，要想获得一个成功的设计，必须先了解任务之间的依赖关系。

1.2.3 数据的可扩展共享

不同任务之间通常需要共享数据。问题是，当一个程序并行运行时，程序的不同部分之间可能会相互竞争着对同一内存位置中存储的数据进行更新。这种预料之外的数据竞争可能是灾难性的。数据竞争问题的解决方案包括同步线程技术。

你可能已经非常熟悉在特定情况下通过阻塞并发线程的执行来实现并发线程同步的技术，例如锁、原子级别的“比较并交换”操作，以及信号灯等。所有这些技术都能达到顺序访问共享资源的效果。对于数据共享，虽然你的第一反应可能是添加锁或其他同步机制，但是添加同步机制降低了应用程序的并行性。任何一种同步形式都是顺序的形式。任务可以以争夺锁结束，而不是做那些你希望它们做的工作。另外，在编程中使用锁也容易出错。

更多关于并行编程中不可变类型的重要性，请参见附录 A 的 A.4 节。

幸运的是，有一些技术允许共享数据，同时又不降低性能或使程序容易出错。这些技术包括：使用不可修改的只读数据；限制程序对共享变量的依赖；在算法中引入新步骤，在适当的检查点合并对可变状态的本地描述。可扩展共享技术可能会牵涉到对现有算法的更改。

可扩展共享可能会牵涉到对现有算法进行修改。

添加同步(锁)会降低应用程序的可扩展性。

传统的面向对象设计可以有复杂的并且高度相关联的对象引用存储器内图。因此,传统的面向对象编程风格可能很难适应可扩展并行的执行。你的第一反应可能就是,把大的相互关联的对象图的各个域作为可变的共享状态,只要多个任务有可能共享这些域,就把对这些域的访问封装在顺序锁里面。遗憾的是,这不是一个可扩展的共享方法。锁通常会对所有内核的性能产生负面影响。锁强制内核停下来相互通信,这需要时间,且在代码中引入串行块,这降低了潜在并行性。内核数量变大时,争用锁的成本就增加。随着共享同一数据的任务数的增加,与锁相关的开销就在计算中占支配地位。

除了性能问题外,依赖于复杂同步的程序还容易发生各种其他问题,包括死锁。当两个或多个任务都在等待对方释放锁时,就会发生死锁。并行编程大多数惨剧的发生实际上都是因为对共享可变状态或者封锁协议的不正确使用。

然而,如果有限制地使用共享状态或者封锁协议,对象图中的同步元素在可扩展并行程序中就是合理的。本书中有节制地使用了同步机制。你也应该这样做。锁可以被看作并行编程的 goto 语句:它们容易出错,但在某些情况下是有必要的。如果编译器和库允许,最好能够保留它们。

不能以性能的名义忽略同步,同步对于正确性来说是必要的。代码首先必须是正确的。然而,在设计过程中还是要秉承少用同步的原则,不要在事后才向应用程序添加同步机制。

1.2.4 设计方法

对于开发者来说,通常是首先确定一个问题的领域,然后并行化代码以提高性能,接着为下一个瓶颈重复这一过程。并行化现有的顺序应用程序时,这是一个特别诱人的方法。虽然这样做可能会提高一些性能,但存在许多陷阱,且产生的结果可能并不是最理想的。还有一个更好的办法,就是先了解你的问题或应用程序,然后把整个应用程序作为一个整体来寻

找潜在的并行性。根据自己的发现来指引自己采用一种不同的体系结构或算法，这个结构或算法更好地显露出应用程序的潜在并行性。不要只是认识到瓶颈，并行化它们。要进行结构调整，为程序做好执行并行操作的准备。

要从数据结构和算法方面思考问题，不要只停留于认识到瓶颈。

分解、协调和可扩展共享技术是相互关联的。它们之间循环引用。当为一个特定的应用程序选择方法时，需要综合考虑这些方面。

看完前面的描述，你可能会抱怨这一切都太含糊了。如何具体地把问题划分成任务？究竟应该使用哪些协调技术？

本书描述的模式就是这些问题的最佳答案。模式是了解这些问题的真正捷径。当你开始查看模式背后的设计动机时，也会形成如何把模式及其变形应用到自己的应用程序中的直觉。下一节将提供更多关于如何选择恰当模式的细节。

使用模式。

1.3 选择恰当的模式

通过表 1.1 来选择相关的模式。

表 1.1 选择恰当的模式

| 应用程序特点 | 相关的模式 |
|----------------------------------------|-------------------------------------------------------------------------|
| 是否有顺序循环，每次迭代的步骤之间是否存在相互通信？ | 并行循环模式(第 2 章) 并行循环在同时有多个输入时，采用独立操作方式。 |
| 不同的操作之间是否存在界限清楚的控制依赖？这些操作大部分都没有顺序依赖关系？ | 并行任务模式(第 3 章) 并行任务可以以分支和合并的方式，建立并行控制流。 |
| 是否需要应用某种组合算子汇总数据？在不完全独立的步骤中是否存在循环？ | 并行合并模式(第 4 章) 并行合并合并部分结果的算法中引入特殊的步骤。这个模式表示一种归约操作，并且包含了映射/简化，作为其变体之一。 |

续表

| 应用程序特点 | 相关的模式 |
|---------------------------------------------|-----------------------------------------------------------------------|
| 算法中步骤的次序是否依赖于数据流限制? | future 模式(第 5 章) future 明确了任务间数据流的依赖关系。这种模式也称为任务图模式。 |
| 算法能否在运行过程中动态划分问题域? 你是在递归数据结构(例如, 图)中进行操作的吗? | 动态任务并行模式(第 6 章) 这个模式采用分而治之的方法, 根据需要产生新任务。 |
| 应用程序重复执行一系列操作吗? 输入数据是否有流的特点? 处理的顺序重要吗? | 流水线模式(第 7 章) 流水线包括以生产者和消费者的形式通过队列连接起来的部件。即使需要考虑输入的顺序, 所有的部件也能并行运行。 |

熟悉六种模式的可行性的一种方法就是, 阅读每一章的第一二页。它为你提供已经证实了的在多种应用程序中起作用的方法概述。然后回过头去更深入地研究可能适用于你的情况的模式。

1.4 关于术语

在其他地方, 你经常会看到并行性和并发性互为同义词使用。但是在本书中这两个词并不相同。

并发性是与多任务和异步输入输出(I/O)相关的概念。它通常是指存在多个执行中的线程, 每个线程获得一个时间片来执行, 时间片用完后由另一个线程获得一个时间片。为了使程序能够对外部刺激(如输入、设备和传感器)做出反应, 并发是必要的。由于其本身的性质, 即使是在只有一个内核的机器中, 操作系统和游戏也是并发的。

由于并行性, 并发线程同一时间在多个内核上同时执行。并行编程的重点是使用多个处理器资源, 当多个内核可用时不再需要经常性地中断程序的执行, 这提高了应用程序的性能。

并发性和并行性的目的是不同的。并发的主要目的是减少时延，防止长时间过去了，每个无阻塞线程都没执行一些计算。换句话说，并发性的目的是防止线程饥饿。

并发是必需的操作。例如，在单核计算机图形用户界面的操作系统中，如果一次有多于一个窗口要更新它的显示区域，则系统必须支持并发。另一方面，并行性仅仅是关于吞吐量的。它是一个优化手段，而不是功能性要求。它的目的是在所有可用内核中充分利用处理器；要做到这一点，它使用非抢占式调度算法，如对要做的工作中的队列或栈的处理算法。

1.5 并行性的极限

被称为阿姆达尔(Amdahl)定律的理论结果指出，并行性带来的性能提高程度受限于应用程序中顺序处理的量。乍看，这有悖于常理。

阿姆达尔定律表明，不论有多少内核，能获得的最大加速比是(1/顺序处理所花费的时间百分比)。图 1.1 说明了这一点。

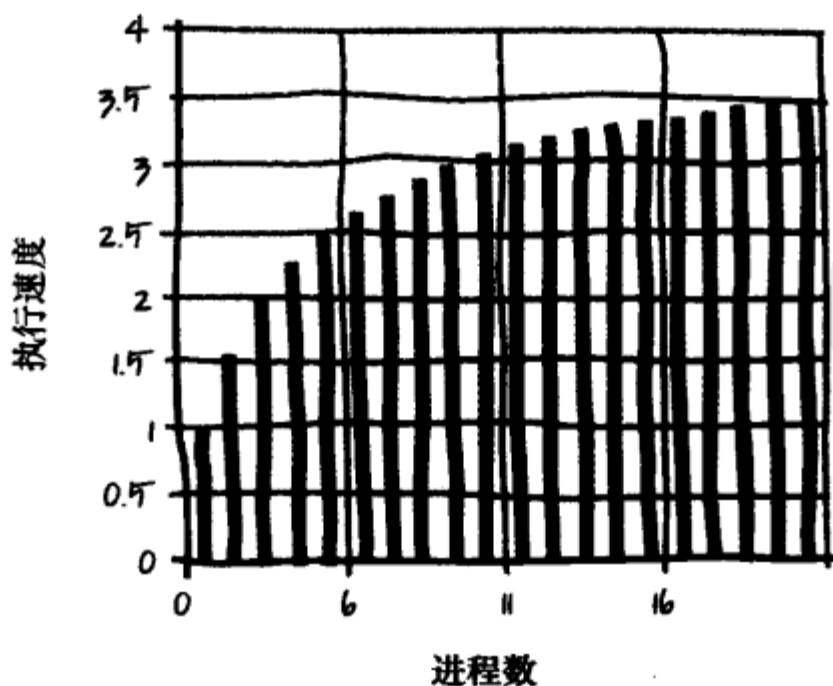


图 1.1 在顺序处理占 25%的应用程序中，阿姆达尔定律的应用

例如，在有 11 个处理器的情况下，应用程序运行速度略超过完全顺序执行的 3 倍。

即使只有较少内核，可以看到预期的加速比也不是线性的。图 1.2 说明了这一点。

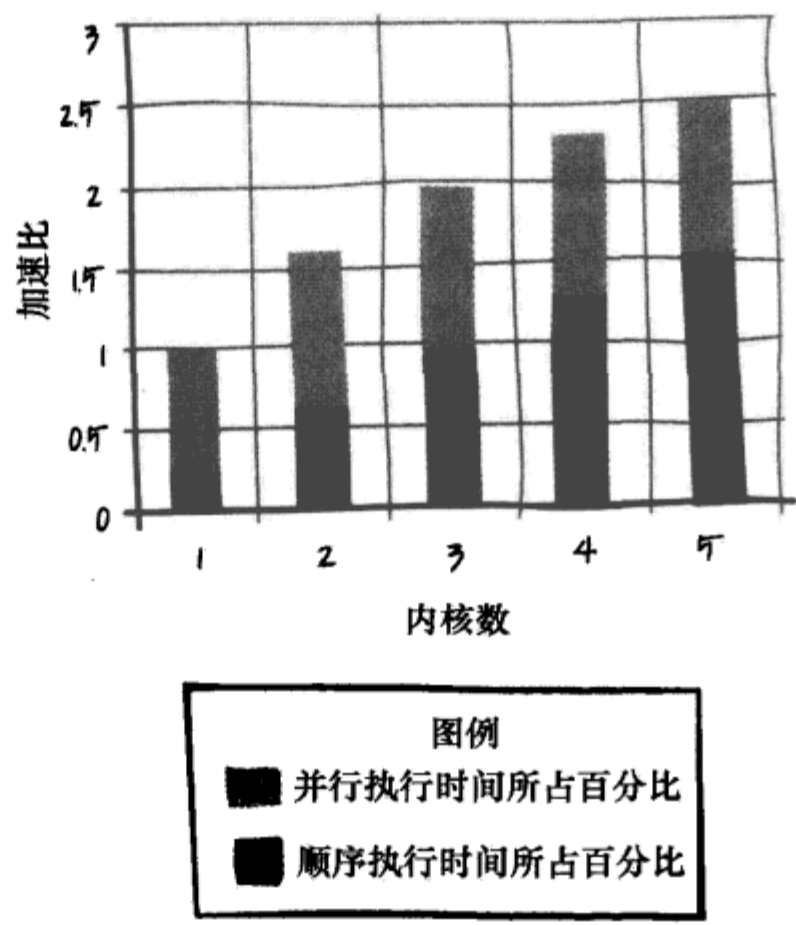


图 1.2 在顺序处理占 25%的应用程序中，每个内核的性能提高

图 1.2 表明随着内核数目(和整个应用程序的速度)的增加，花费在应用程序中顺序部分的时间百分比就增加(花费在顺序处理上的时间是不会改变的)。该图也显示了相对于示例程序，为什么在实际应用中，你可能满足于四核电脑上的 2 倍加速比。应用程序的可扩展性永远是重要的问题。可扩展性取决于花费在本质上顺序进行的工作上的时间量。

阿姆达尔定律的另一个含义是：对于某些问题，你可能想要在应用程序中可并行执行的部分创建附加功能。例如，电脑游戏的开发人员可能会发现，在较新的多核电脑上使用并行硬件，制作出日益复杂的图形是可能

的，即便可行性不如让游戏在逻辑上并行。性能会影响应用程序的功能组合。

在实际中获得的加速比通常要小于阿姆达尔定律所预期的。随着内核数的增加，由访问共享内存带来的开销也增加。此外，并行算法可能包括协调带来的开销，而这在顺序情况下是不必要的。如 Visual Studio Concurrency Visualizer 这样的分析工具可以帮助你了解并行性有多高效。

总之，因为应用程序包含了一部分必须顺序运行，一部分可以并行运行的代码，所以，总的来说，随着内核数量的线性增加，即便应用程序的某些部分看似呈线性加速，应用程序的性能也很难会线性提高。了解应用程序的架构和它的算法——也就是说，应用程序的哪些部分适合于并行执行——是分析性能必不可少的一个步骤。

1.6 一些技巧

总是要争取使用最简单的方法。以下是一些基本原则。

- 只要有可能，就要停留在抽象的最高层，使用构造块或库去做并行工作。
- 使用应用程序服务器固有的并行性，如使用纳入 Web 服务器或数据库中的并行性。
- 使用 API 封装并行性，如 .NET Framework 下的微软并行扩展 (TPL 和 PLINQ)。这些类库是由专家撰写的，并已全面通过测试，它们能帮助你避免许多并行编程常见的问题。
- 当思考如何并行化应用程序时，应考虑应用程序的总体架构。人们很容易只寻找性能热点，然后集中研究如何提高性能。虽然这可能会提高性能，但它不一定能给你最好的结果。

- 使用模式，如本书描述的各种模式。
- 通常情况下，调整算法(例如，消除对共享数据的需要)比在底层改进那些原本设计为顺序运行的代码要好得多。
- 除非绝对需要，否则并发任务之间不要共享数据。如果共享了数据，就使用现有 API 提供的其中一个容器，如共享队列。
- 束手无策时才使用低级别原语，如线程和锁。从线程到任务，提高应用程序的抽象水平。

1.7 练习

1. 一个问题是分解成许多小任务，还是分解成较大一些的任务，如何权衡？
2. 当把一个程序从单核计算机移植到四核计算机时，它花费在顺序处理的时间是在单核系统的 10%，程序的最大潜在加速比是多少？
3. 并行性和并发性有何不同？

1.8 扩展阅读

如果有兴趣进一步了解本书中的术语，请参考本书末尾的“术语表”。

本书中介绍的设计模式与工业界和学术界的组织所制定的并行模式分类是一致的。在这些组织的术语体系中，本书中的模式可以被认为是算法或实现模式。并行模式的分类方法，我们可以从 Mattson 等人的书中和我们的模式语言(OPL)网站上找到。本书的术语尽量与这些来源的术语相一致。在术语不一致的地方，文中都有注释。

如果需要 Windows 平台上并行性的详细讨论，请参见 Duffy 的书。 .NET 的线程和同步的概述可以在 Albahari 的书中找到。

- Albahari J, Albahari B. *C# 4 in a Nutshell*. fourth edition. O'Reilly, 2010.
- Duffy J. *Concurrent Programming on Windows*. Addison-Wesley, 2008.
- Mattson T G, Sanders B A, Massingill B L. *Patterns for Parallel Programming*. Addison-Wesley, 2004.
- Our Pattern Language for Parallel Programming Ver 2.0.
<http://parlab.eecs.berkeley.edu/wiki/patterns>.





第2章 并行循环

当需要对集合中的每个元素执行相同的独立操作,或者需要进行固定数量的迭代时,就用并行循环模式吧!如果它们不写入由其他步骤读取的内存或者文件,那么这个循环的所有步骤都是相互独立的。

并行循环的语法与你已经知道的 `foreach` 循环很相似,但是,在有可用内核的计算机上,并行循环运行速度会更快。另一个不同点就是,并行循环不会像顺序循环那样限定执行的顺序。各个步骤常常同时进行,即是并行的。有时候,两个步骤会按照与在顺序循环中相反的顺序执行。唯一可以保证的就是,所有的循环迭代一直运行,直到循环结束。

将一个顺序循环转变成一个并行循环是很简单的。然而,我们也很容易将并行循环用错地方。这是因为很难确定这些步骤实际上是否是相互独立的。当一个步骤与另一个步骤有关联时,我们需要通过实践来学会如何识别这种情况。有时候,在一个各个步骤都有相互依赖关系的循环中,使用这种模式会促使这些程序以一种完全意想不到的方式运行,而且可能会停止响应。有时候,并行循环模式会引起一个发生几率为百万分之一的潜在错误。换句话说,“独立”一词是并行模式定义的重要组成部分,而且,这就是本章要详细介绍的一部分。

独立的并行循环模式适用于多数数据元素的操作。它也是数据并行的一个例子。

对于并行循环,并行度不需要由代码指定。实时运行环境会以尽可能多的内核来同时执行循环步骤。无论有多少内核可用,循环都能正常运行。如果只有一个内核,其性能接近于顺序循环的性能(也许相差无几)。如果有多个内核,性能会提高;在大多数情况下,性能的提高与内核的数量是成正比的。

2.1 基础知识

为了使具有独立迭代性质的 for 和 foreach 循环在多核计算机中运行得更快，请使用并行形式。

.NET 体系既包括并行 For 循环，也包括并行 ForEach 循环，并且在并行 LINQ(PLINQ)查询语言中也是有效的。可以使用 Parallel.For 方法去遍历一个整数索引范围，也可以使用 Parallel.ForEach 方法去遍历用户提供的值。如果喜欢用一个高层次的、声明式的表现形式去描述循环，或者，想充分利用 PLINQ 的便利性和灵活性的优势，那么就使用 PLINQ。

2.1.1 并行 for 循环

下面是一个用 C#编写的顺序 for 循环的例子。

```
int n = ...
for (int i = 0; i < n; i++)
{
    // ...
}
```

记住，如果想使用一个并行循环，那么循环体的各个步骤必须是相互独立的。这些步骤不能通过共享变量相互联系。

为了利用多核的优势，通过调用 Parallel.For 方法来更换 for 循环的关键字，并将循环体转换成一个 lambda 表达式。

```
int n = ...
Parallel.For(0, n, i =>
{
    // ...
});
```

Parallel.For 使用多核针对特定索引范围进行操作。

Parallel.For 是一个重载的静态方法。下面是例子中用到的 Parallel.For 方法的声明。

```
Parallel.For(int fromInclusive,
             int toExclusive,
             Action<int> body);
```

本例中，前两个参数指定了迭代的限制。第一个参数是循环的最低索引。第二个参数是上限值，或者说是最大索引值加一。第三个参数就是每次迭代都要调用一次的操作。这个操作将迭代索引作为它的参数，对于每个索引，循环体都运行一次。

Parallel.For 方法还有其他重载版本。在 2.3 节和第 4 章中都有所涉及。

该示例包括形式为“args \Rightarrow body”的 lambda 表达式，而它正是 **Parallel.For** 方法的第三个参数。lambda 表达式是一种能从它的闭合范围内捕获变量的匿名方法。当然，body 参数也可能是一个委托型的实例、一个匿名方法(使用 **delegate** 关键字)或一个普通命名方法。换句话说，如果不需要，不必使用 lambda 表达式。本书中的例子使用 lambda 表达式，因为它们使代码位于循环体内，并且，当代码行数比较少时，就很容易读懂。

2.1.2 并行 Foreach 循环

下面是一个用 C# 编写的顺序 **foreach** 循环的例子。

```
IEnumerable<MyObject> myEnumerable = ...

foreach (var obj in myEnumerable)
{
    // ...
}
```

为了利用多核的优势，将 **foreach** 关键字替换为对 **Parallel.Foreach** 方法的调用。

```
IEnumerable<MyObject> myEnumerable = ...

Parallel.Foreach(myEnumerable, obj =>
{
    // ...
});
```

Parallel.For 方法不能保证任何特定顺序的执行。不像一个顺序循环，一些高值的索引也许会在一些低值的索引之前被处理。

如果对 lambda 表达式的语法不熟悉，请看 2.8 节“扩展阅读”推荐的参考资料。体验 lambda 表达式后，就会爱不释手。

`Parallel.ForEach` 运行集合中每一个元素的循环体。

不要忘记，迭代必须是相互独立的。循环体中只能更新那些传递给循环体的特定实例的字段。

可以通过运用 `AsParallel` 扩展方法，将 LINQ 表达式转换为并行代码。

`Parallel.ForEach` 是一个重载的静态方法。下面是例子中用到的 `Parallel.ForEach` 方法的声明。

```
ForEach<TSource>(IEnumerable<TSource> source,
                 Action<TSource> body);
```

例子中，第一个参数是一个实现 `IEnumerable<MyObject>` 接口的对象。第二个参数是输入集合中每个元素都要调用的方法。

`Parallel.ForEach` 方法不保证顺序执行。不像顺序的 `ForEach` 循环，输入值并不总是顺序执行的。

`Parallel.ForEach` 方法还有其他重载版本，在 2.3 节和第 4 章中都有所涉及。

2.1.3 并行 LINQ(PLINQ)

.NET Framework 的语言集成查询(LINQ)的功能包括一个叫做 PLINQ 的并行版本(并行 PLINQ)。为表达 PLINQ 查询，可采用多种方法和形式，但是，通过增加一个对 `AsParallel` 扩展方法的调用，几乎所有的 LINQ-to-Objects 表达式都很容易转化为对应的并行版本。以下是一个同时显示了 LINQ 版本和 PLINQ 版本的例子。

```
IEnumerable<MyObject> source = ...

// LINQ
var query1 = from i in source select Normalize(i);

// PLINQ
var query2 = from i in source.AsParallel()
              select Normalize(i);
```

此代码示例创建了两个变换枚举对象 `source` 的查询。如果有多个内核可用，PLINQ 版本就使用多个内核。

在欲遍历所有输入值而不需要选择返回的输出值时，也可以使用 PLINQ 版本的 ForAll 表达式方法。以下代码呈现了这种情况。

```
IEnumerable<MyObject> myEnumerable = ...  
  
myEnumerable.AsParallel().ForAll(obj => DoWork(obj));
```

ForAll 扩展方法等同于 PLINQ 中的 Parallel.ForEach 方法。

2.1.4 预期

默认情况下，并行度（也就是说，在硬件上，有多少次迭代能在同一时间运行）取决于可用内核的数量。特殊情况下，内核越多，循环运行的速度就越快，直到达到由阿姆达尔定律预测的收益递减点。循环能运行多快取决于循环类型。

并行循环模式的 .NET 实现确保了在循环体运行时抛出的异常不会丢失。对于 Parallel.For 和 Parallel.ForEach 这两种方法以及 PLINQ，异常会被收集到 AggregateException 对象中，在调用线程的情况下重新抛出。所有的异常会传回给你。要了解更多的有关并行循环的异常处理，请阅读 2.3 节。

并行循环有很多变化形式。Parallel.For 有 12 种重载方法，Parallel.ForEach 有 20 种重载方法。PLINQ 有将近 200 个扩展方法。虽然有多种 For 和 ForEach 的重载版本，可以将重载当成提供可选配置的选项。其中有两个重载方法和最大并行度和外部中断挂钩。这些选项允许循环体去监视其他步骤的进展（例如，去看是否有异常在等待）和管理本地任务状态。在高级方案中，这些选项有时候是必需的。要了解最重要的案例，请阅读 2.3 节。

如果将顺序循环转化成并行循环，随后发现程序没有按预期运行，最有可能的问题就是循环的步骤不是相互独立的。以下是循环体依赖的一些

用 PLINQ 版本的 ForAll 表达式方法来替代被当成 Parallel.ForEach 方法参数的 PLINQ 查询是很重要的。有关详细信息，请查阅 2.4.6 节。

增加内核使循环运行得更快；然而，内核数量总会有一个上限值。

必须选择正确的粒度。很多的小并行循环可以达到过度分解的点，这种情况下，多核加速效果会被并行循环开销所抵消。

完善的异常处理机制是并行循环处理的重要一环。

仔细检查循环迭代之间的依赖关系！不注意各步骤之间的依赖关系是目前为止在并行循环中产生的最常见的错误。

常见例子。

- **写共享变量。**如果循环体中写入一个共享变量，那么就有一个循环体依赖。最常见的情况是当你合计总值时。以下是一个例子，其中 `total` 为各次迭代共享。

```
for(int i = 1; i < n; i++)  
    total += data[i];
```

如果遇到这种情况，请参阅第 4 章。

共享变量有不同风格。任何在循环体范围之外声明的变量都是一个共享变量。类或数组等类型的共享引用允许所有字段或数组元素的共享。使用 `ref` 关键字声明的参数就成为了共享变量。即使读取和写入文件也会有和共享变量一样的效果。

- **使用对象模型的属性。**如果循环体处理的对象公开属性，需要知道这些属性适用于共享状态还是只针对于对象本身的局部状态。例如，称为 `Parent` 的属性可能适用于全局的状态。示例如下。

```
for(int i = 0; i < n; i++)  
    SomeObject[i].Parent.Update();
```

本例中，迭代循环可能不是独立的。对于 `i` 的所有值，`SomeObject[i].Parent` 是对一个单一共享对象的引用。

- **引用不是线程安全的数据类型。**如果并行循环体使用一个线程不安全的数据类型，那么，此循环体就不是独立的(在线程中有一个隐含的依赖关系)。这方面的一个例子以及相应的解决方案将在 2.3.6 节中呈现。
- **循环携带的依赖。**如果并行 `for` 循环体的计算与循环索引有关，就有可能产生循环携带的依赖。以下代码将体现出这种依赖。循环体中引用了 `data[i]` 和 `data[i - 1]`。如果这里用到 `Parallel.For`,

从属性和方法获得数据时，必须格外小心。大型对象模型是因为用一种难以置信的迂回方式来共享可变状态而被人们所知。

也不能保证在循环体中，对 `data[i]` 执行循环之前已经更新了 `data[i - 1]`。

```
for(int i = 1; i < N; i++)  
    data[i] = data[i] + data[i - 1];
```

有时候，在循环携带依赖的例子中，有可能会用到并行算法，但是，这超出了本书的范围。最好的选择就是在程序的其他地方寻找并行的机会，或是分析算法，看它是否匹配一些在科学计算中出现的先进并行模式。并行扫描和并行动态规划就是这种模式的例子。

对循环指针变量进行运算，特别是加法或减法，通常产生循环携带依赖。

当寻找并行的机会时，分析应用程序是一种很好的方法，它可以让你更加了解应用程序在哪些地方花费了时间。然而，分析并不能替代对应用程序的结构和算法的理解。例如，分析并不能告诉你循环体是否是相互独立的。

不要期待从分析中获得奇迹，它不能为你分析算法。只有你能分析它。

2.2 示例

以下是一个何时使用并行循环的例子。Fabrikam Shipping 公司准备扩大它的商业账户的信贷。它利用客户的信用情况去确定可能造成风险的账户。每个客户账户包括过去的结欠余额的历史记录。Fabrikam 已经注意到，在他们拖欠款项之前的数月中，那些不支付自己账单的客户往往有稳步增长余额的历史记录。

为确定风险账户，Fabrikam 使用统计趋势分析，对每个账户计算其预计的贷方余额。如果分析预测一个客户账户将在三个月内超过其信用额度，该账户将标记为需要 Fabrikam 的信用分析师人工审核。

在该应用程序中，顶层循环在账户存储库中对客户进行遍历。循环体设置了结余历史的趋势线，再预计出资产负债表，将其与信用额度进行比

较，必要时设置警告标志。

此应用的一个重要方面是每一个客户的信用状态可以独立计算。每个客户的信用状态不依赖于其他客户的信用状态。因为操作是独立的，只需将顺序 `foreach` 循环替换成并行循环，就可以使信用分析程序运行得更快。

这个例子的完整源代码可从网上获取，网址为 <http://parallelpatterns.codeplex.com>，目录为 `Chapter2\CreditReview`。

2.2.1 信贷审查的顺序版本示例

以下是一个信贷分析操作的顺序版本。

```
static void UpdatePredictionsSequential(  
    AccountRepository accounts)  
{  
    foreach (Account account in accounts.AllAccounts)  
    {  
        Trend trend = SampleUtilities.Fit(account.Balance);  
        double prediction = trend.Predict(  
            account.Balance.Length + NumberOfMonths);  
        account.SeqPrediction = prediction;  
        account.SeqWarning = prediction < account.Overdraft;  
    }  
}
```

`UpdatePredictionsSequential` 方法处理应用程序账户库中的每一个账户。`Fit` 方法是一种使用最小二乘统计法从数字数组中创建一条趋势线的实用工具函数。`Fit` 方法是一个纯函数。这也意味着它不修改任何状态。

该预测在为期三个月的预测趋势基础上进行。如果预测超过透支额度（信贷余额在账户系统中是负值），这个账户被标记为审查。

2.2.2 使用 Parallel.ForEach 的信贷审查示例

信用评分分析的并行版本与顺序版本很相似。

```
static void UpdatePredictionsParallel(AccountRepository
accounts)
{
    Parallel.ForEach(accounts.AllAccounts, account =>
    {
        Trend trend = SampleUtilities.Fit(account.Balance);
        double prediction = trend.Predict(
            account.Balance.Length + NumberOfMonths);
        account.ParPrediction = prediction;
        account.ParWarning = prediction < account.Overdraft;
    });
}
```

`UpdatePredictionsParallel` 方法中，除了用 `Parallel.ForEach` 方法替换 `foreach` 操作，其他的都等同于 `UpdatePredictionsSequential` 方法。

2.2.3 PLINQ 信贷审查示例

可以用 PLINQ 来表示一个并行循环，示例如下。

```
static void UpdatePredictionsPlinq(AccountRepository
accounts)
{
    accounts.AllAccounts
        .AsParallel()
        .ForAll(account =>
        {
            Trend trend = SampleUtilities.Fit(account.Balance);
            double prediction = trend.Predict(
                account.Balance.Length + NumberOfMonths);
        });
}
```

```
account.PlinqPrediction = prediction;  
account.PlinqWarning = prediction < account.Overdraft;  
});  
}
```

使用 PLINQ 几乎与使用 LINQ-to-Objects 完全一样。PLINQ 提供了一个 `ParallelEnumerable` 类，它定义了不同类型的可扩展方法，其方式非常类似于 LINQ 中的 `Enumerable` 类。`ParallelEnumerable` 中的其中一个方法就是 `AsParallel` 扩展方法。

`AsParallel` 扩展方法允许将 `IEnumerable<T>` 类型的顺序集合转换成一个 `ParallelQuery<T>` 对象。将 `AsParallel` 应用到 `accounts.AllAccounts` 集合，会返回一个 `ParallelQuery<AccountRecord>` 类型的对象。

PLINQ 的 `ParallelEnumerable` 类有近 200 个为 `ParallelQuery<T>` 对象提供并行查询的扩展方法。除了 LINQ 方法的并行实现，例如 `Select` 和 `Where`，PLINQ 还提供一个 `ForAll` 扩展方法，此方法为每个元素调用一个并行的委托方法。

在 PLINQ 预测示例中，`ForAll` 的参数是一个对指定账户进行信用分析的 `lambda` 表达式，而其中的循环体与顺序版本中的循环体是一样的。

2.2.4 性能比较

在四核计算机中运行此信贷审查示例可以看出，`Parallel.ForEach` 和 PLINQ 版本的运行速度将近达到了顺序版本的四倍。在不同机器上运行加速效果有差异。可以在自己的计算机上尝试运行在线样本。

2.3 变化形式

信贷分析实例展示了一个运用并行循环的典型方法，但是可能会有变化。本节中介绍了一些最重要的信息。不需要总是使用这些变化形式，但

是应该知道它们是可用的。

2.3.1 尽早中断循环

使用 `break` 中断循环是顺序迭代中很熟悉的一部分。在并行循环中，它不太常见，但是，有时候需要这么做。以下是一个顺序情况下的例子。

```
int n = ...
for (int i = 0; i < n; i++)
{
    // ...
    if (/* stopping condition is true */)
        break;
}
```

并行循环中的这种情况更复杂，因为有多个步骤在同一时间段运行，而且，并行循环中的各个步骤不一定按预先确定的顺序运行。因此，并行循环有两种方法去中断或停止循环。终止循环前，并行中断允许完成当前迭代之前的所有线程上的所有迭代。而并行停止则立即终止所有的迭代。

1. 并行中断

`Parallel.For` 方法有一个重载方法，此重载方法提供一个 `ParallelLoopState` 对象，此对象常作为循环体的第二个参数。可以通过调用 `ParallelLoopState` 对象中的 `Break` 方法来请求中断循环。以下就是这样一个例子。

```
int n = ...
Parallel.For(0, n, (i, loopState) =>
{
    // ...
    if (/* stopping condition is true */)
    {
        loopState.Break();
    }
})
```

使用 `Break` 提前退出循环，同时确保低索引步骤完成。

```
        return;  
    }  
});
```

这个例子使用 `Parallel.For` 的重载版本,它可以将一个 `loop state`(循环状态)对象传递到每一步。以下是示例中用到的 `Parallel.For` 方法的声明。

```
Parallel.For(int fromInclusive,  
             int toExclusive,  
             Action<int, ParallelLoopState> body);
```

传递给 `loopState` 参数的对象是 `ParallelLoopState` 类的一个实例, `ParallelLoopState` 类是通过在循环体内使用并行循环来构建的。

调用 `ParallelLoopState` 对象的 `Break` 方法后,就开始有序地停止循环操作。调用 `Break` 时任何正在运行的步骤都将继续运行直到完成。

在长时间运行的循环体中,可能需要检查中断条件;如果有中断请求,则立即退出该步骤。如果不这么做,就将继续运行直到结束。要查看并行运行中的另一个步骤是否也有中断请求,可以通过检索并行循环状态的 `LowestBreakIteration` 属性的值来得出结论。如果它返回了一个可为空的长整数,且其 `HasValue` 属性为 `true`,就知道有中断请求了。也可以读取循环状态对象的 `ShouldExitCurrentIteration` 属性,就像读取其他终止条件一样检测出中断。

在调用 `Break` 方法的过程中,当迭代的索引值比当前值小时,那么就开始运行该迭代(如果它们还没开始运行),但是,索引值大于当前值的迭代就不允许其运行。这确保了所有低于中断点的迭代都将完整运行。

由于并行执行,有可能不止一个步骤被暂停。在这种情况下,中断结束后,索引值最低的步骤将最先开始运行。

`Parallel.For` 和 `Parallel.ForEach` 方法将返回一个 `ParallelLoopResult` 类型的对象。通过检查两个循环结果属性的值,可以找出是否以中断的方

调用 `Break` 不会停止其他已经开始运行的步骤。

不要忘记,即使在调用 `Break` 之后,所有比调用 `Break` 方法的步骤所拥有的索引值小的步骤将被允许正常运行。

式终止运行。如果 `IsCompleted` 属性为 `false` 且 `LowestBreakIteration` 属性返回一个 `HasValue` 属性为 `true` 的对象,那么,就知道循环是通过调用 `Break` 方法来终止循环的。可以利用循环结果的 `LowestBreakIteration` 属性来查询具体的索引值。下面就是这样一个例子。

```
int n = ...
var result = new double[n];

var loopResult = Parallel.For(0, n, (i, loopState) =>
{
    if (/* break condition is true */)
    {
        loopState.Break();
        return;
    }
    result[i] = DoWork(i);
});

if (!loopResult.IsCompleted &&
    loopResult.LowestBreakIteration.HasValue)
{
    Console.WriteLine("Loop encountered a break at {0}",
        loopResult.LowestBreakIteration.Value);
}
```

`Break` 方法可以确保所有比某个特定迭代索引值低的数据都被很好地处理。根据迭代进行的方式可以看出,比调用 `Break` 方法的步骤的索引值还高的那些步骤有可能在调用 `Break` 方法之前已经开始执行。

`Parallel.ForEach` 方法也支持循环状态的 `Break` 方法。因为并行循环会以枚举输入的方式进行排序,所以它会从零开始分配项目序号。这个序号用作 `LowestBreakIteration` 属性的迭代索引。

2. 并行停止

还有一些情况下,你希望停止条件一旦达到,循环就尽可能快地停止,

注意,比调用 `Break` 方法的步骤的索引值还高的那些步骤可能已经运行。但是,没有办法预测它何时开始运行或者它是否已经开始运行。

`Parallel.ForEach` 方法也支持循环状态的 `Break` 方法。

如果不需要所有低索引迭代在循环终止之前运行，就可以使用 **Stop** 方法尽早退出循环。

例如在无序搜索中。“中断”和“停止”之间的区别是这样的：对于停止，如果尚未运行，就不会尝试执行迭代索引低于停止索引的循环。要以这种方式停止一个循环，可以调用 **ParallelLoopState** 类的 **Stop** 方法，而不是 **Break** 方法。下面是一个并行停止的例子。

```
var n = ...
var loopResult = Parallel.For(0, n, (i, loopState) =>
{
    if (/* stopping condition is true */)
    {
        loopState.Stop();
        return;
    }
    result[i] = DoWork(i);
});

if (!loopResult.IsCompleted &&
    !loopResult.LowestBreakIteration.HasValue)
{
    Console.WriteLine("Loop was stopped");
}
```

当 **Stop** 方法被调用时，导致停止的那次迭代的索引值是不可用的。

在同一个并行循环中，不能同时调用 **Break** 方法和 **Stop** 方法，只能从这两个循环退出行为中选出想用的那个。如果同时调用 **Break** 方法和 **Stop** 方法，将抛出一个异常。

并行程序中更常用的是 **Stop** 方法而不是 **Break** 方法。当循环体本身是相互独立的时候，处理所有比停止迭代的索引还小的迭代通常是没有必要的。还有一个事实就是，**Stop** 停止一个循环比 **Break** 要快很多。

PLINQ 查询中不存在 **Stop** 方法，但是，可以用 **WithCancellation** 扩展方法，随后再使用取消操作来终止 PLINQ 查询。欲了解更多信息，请阅读 2.3.2 节。

你可能会更常用 **Stop** 而不是 **Break**。

2.3.2 外部循环取消

在某些情况下，由于外部请求，可能需要取消一个并行循环。例如，可能需要停止正在做的去响应一个来自用户接口的请求。

在 .NET 中，可以用 `CancellationTokenSource` 类去表示取消，用 `CancellationToken` 结构去检测和响应一个取消请求。该结构能够找出是否有挂起的请求。该类能够让你知道将要发生取消操作。

`Parallel.For` 和 `Parallel.ForEach` 方法包括了允许将并行循环选择作为一个参数的重载版本。可以将一个取消标记指定为其中一个选项。如果提供一个作为并行循环选项的取消标记，循环将用这个标记去查找取消请求。示例如下。

```
void DoLoop(CancellationTokenSource cts)
{
    int n = ...
    CancellationToken token = cts.Token;

    var options = new ParallelOptions
        { CancellationToken = token };
    try
    {
        Parallel.For(0, n, options, (i) =>
        {
            // ...

            // ... optionally check to see if cancellation happened
            if (token.IsCancellationRequested)
            {
                // ... optionally exit this iteration early
                return;
            }
        })
    }
```



```

    });
}
catch (OperationCanceledException ex)
{
    // ... handle the loop cancellation
}
}

```

以下是本例中用到的 `Parallel.For` 方法的声明。

```

Parallel.For(int fromInclusive,
             int toExclusive,
             ParallelOptions parallelOptions,
             Action<int> body);

```

外部取消需要一个取消标记源对象。

当 `DoLoop` 方法的调用者准备好取消时，将调用 `CancellationTokenSource` 类的 `Cancel` 方法，并作为参数提供给 `DoLoop` 方法。并行循环将完成当前正在运行的迭代，随后，抛出一个 `OperationCanceledException` (操作取消异常)。取消操作开始后，不再有新的迭代运行。

如果已经告知有外部取消，并且，循环已经调用了 `ParallelLoopState` 对象的 `Break` 方法或 `Stop` 方法，那么这两个方法将会竞争谁先被识别。并行循环要么抛出一个 `OperationCanceledException`，要么就使用在 2.3.1 节中描述过的 `Break` 方法和 `Stop` 方法机制来终止。

可以用 `WithCancellation` 扩展方法去增加 PLINQ 查询的外部取消功能。

2.3.3 异常处理

抛出一个未处理异常会防止新迭代的开始。

如果并行循环体抛出一个未处理异常，并行循环就不能再开始新的步骤。默认情况下，抛出异常时正在执行的迭代，不同于抛出异常的迭代，它将继续执行完毕。当它们完成迭代后，并行循环将在调用它的线程中抛出一个异常。遇到长时间运行的迭代时，可能要测试是否在另一个迭代中还有异常正在等待处理。这可以通过 `ParallelLoopState` 类的 `IsExceptional`

属性来判断。如果有一个异常正在等待处理，这个属性就会返回 `true`。

因为并行执行时可能有多个异常发生，所以要使用异常类型来对异常进行分组，这就是所谓的汇总异常。`AggregateException` 类有一个内部异常属性，这个属性包括了一个在执行并行循环时可能发生的所有异常的集合。由于循环是并行运行的，就可能会有多个异常。

异常优先于外部取消和通过调用 `ParallelLoopState` 对象的 `Break` 或 `Stop` 方法发起的循环终止。

欲了解相关的代码示例和有关汇总异常处理的信息，请阅读 3.3.2 节。

2.3.4 小循环体的特殊处理

如果循环体中只执行少量的操作，你会发现，通过将迭代划分为较大的操作单元，可以获得更好的性能。这样做的原因是，在处理一个循环时会引入两种类型的开销：管理处理线程的成本和调用委托方法的成本。大多数情况下，这些开销可以忽略不计，但是对于小循环体，它们就很明显了。

.NET Framework 4 的并行扩展包括对自定义分区的支持。一个 `Partitioner` 对象将索引划分为一个个不重叠的范围。通过分区程序的作用，每个并行循环步骤处理一个索引范围而不是单个的索引。

如果有许多迭代，这些迭代中的每一个都只执行少量的操作，那么，请考虑使用一个 `Partitioner` (分区)对象。

通过将迭代分组为一个个的范围，可以避免并行循环的一些正常开销。以下就是这样的一个例子。

```
int n = ...
double[] result = new double[n];
Parallel.ForEach(Partitioner.Create(0, n),
    (range) =>
    {
        for (int i = range.Item1; i < range.Item2; i++)
        {
```

```
        // very small, equally sized blocks of work
        result[i] = (double)(i * i);
    }
});
```

以下是上述例子中用到的 `Parallel.For` 方法的声明。

```
Parallel.ForEach<TSource>(
    Partitioner<TSource> source,
    Action<TSource> body);
```

在这个例子中,可以将 `Partitioner.Create` 方法的结果看作一个类似于 `IEnumerable<Tuple<int, int>>` 实例的对象(实际类型是 `Partitioner<Tuple<int, int>>`)。换句话说,通过两个整数字段值,可以创建一个元组集合(未命名记录)。每个元组代表在并行循环的单次迭代中处理的索引值范围。并行循环的每次迭代都包含一个嵌套顺序 `for` 循环,以对该范围内的每个值进行处理。

基于分区的并行循环语法比.NET 中的并行循环语法更复杂,而且,当每次迭代中的操作都很多(或各次迭代大小不均)时,它可能不会带来更好的性能。通常来说,经过分析或是在循环体非常小且迭代次数很大的情况下,只能使用这种更为复杂的语法。

通过 `Partitioner` 对象创建的范围的数量取决于计算机的内核数量。默认的范围数量大约是内核数量的三倍。

如果知道确切的范围,可以用一个允许指定每个范围大小的 `Partitioner.Create` 方法重载版本来实现划分。示例如下。

```
double[] result = new double[1000000];
Parallel.ForEach(Partitioner.Create(0, 1000000, 50000),
    (range) =>
    {
        for (int i = range.Item1; i < range.Item2; i++)
```

```
{  
    // small, equally sized blocks of work  
    result[i] = (double)(i * i);  
}  
});
```

在这个例子中，每个范围涵盖 50 000 个索引值。换句话说，对于百万次迭代，系统将使用 20 次并行迭代(1 000 000/50 000)。这些迭代将分散到所有可用的内核来进行处理。

在并行循环的 API 中，自定义分区是一个扩展点。可以实现自己的分区策略。如需关于这个主题的更多信息，请阅读 2.8 节推荐的参考资料。

2.3.5 控制并行度

虽然通常让系统管理如何将并行循环迭代对应电脑内核，但是，在某些情况下，可能需要额外的控制。

你将看到各种情况下并行循环模式的这种变化。降低并行度往往用于性能测试，以模拟性能较差的硬件。当循环迭代在等待进行输入输出上花费了太多的时间，将并行度增加到比内核数还大的数量就很恰当了。

可以控制并行循环中同时运行的最大线程数。

术语“并行度”有两个用法。最简单的情况下，它指的是同时处理迭代的内核数量。然而，.NET 也用这个术语来指并行循环中同时执行的任务数量。例如，ParallelOptions 对象的 MaxDegreeOfParallelism 属性是指在任意时刻被并行循环调度的操作任务的最大数目。

为了高效地运用硬件资源，任务数量往往比可用的内核数要多。例如，在不需要使用处理器资源的 I/O 操作出现拥塞的情况下，并行循环可能会执行其他的任务。并行度是由系统的基本组件自动管理的；在大多数条件下，Parallel 类的实现、默认任务调度程序和.NET 线程池都能够优化吞吐量。

通过指定 `ParallelOptions` 对象的 `MaxDegreeOfParallelism` 属性, 可以限制同时执行任务的最大数量值。以下就是这样一个例子。

```
var n = ...
var options = new ParallelOptions()
                { MaxDegreeOfParallelism = 2 };
Parallel.For(0, n, options, i =>
{
    // ...
});
```

在前面的代码示例中, 并行循环最多同时执行两个任务。下面是本例中用到的 `Parallel.For` 方法的声明。

```
Parallel.For(int fromInclusive,
             int toExclusive,
             ParallelOptions parallelOptions,
             Action<int> body);
```

通过设置 `ParallelQuery<T>` 对象的 `WithDegreeOfParallelism` 属性, 也可以为 PLINQ 查询设置最大的工作线程数。以下就是这样一个例子。

```
IEnumerable<T> myCollection = // ...

myCollection.AsParallel()
             .WithDegreeOfParallelism(8)
             .ForAll(obj => /* ... */);
```

任何时候, 代码示例中的查询都将以 8 个任务的最大限额来运行。

如果想要指定一个更大的并行度, 可能需要用到 `ThreadPool` 类的 `SetMinThreads` 方法, 以便这些线程被创建时没有延时。否则, 线程池的线程注入算法可能会限制线程加到被并行循环使用的工作线程池中的速度。相比于创建所需数量的线程这项操作, 它将花费更多时间。

2.3.6 在循环体中使用局部任务状态

在执行并行循环时，有时候需要保持局部线程的状态。例如，可能需要使用一个并行循环来初始化随机矩阵中的每个元素。.NET Framework `Random` 类不支持多线程访问。因此，对于每一个线程，都需要一个随机数发生器的单独实例。

以下是一个例子，这个例子使用 `Parallel.ForEach` 方法中的一个重载。因为每一步完成的操作量很少而步骤的数量很大，这个示例就使用一个 `Partitioner` 对象将操作分解成比较大的块。

对于那些调用了线程不安全的方法的循环体，必须使用局部任务状态。

如果循环体需要，局部任务状态是有效的。

```
int numberOfSteps = 10000000;  
double[] result = new double[numberOfSteps];  
  
Parallel.ForEach(  
    Partitioner.Create(0, numberOfSteps),  
    new ParallelOptions(),  
    () => { return new Random(MakeRandomSeed()); },  
    (range, loopState, random) =>  
    {  
        for (int i = range.Item1; i < range.Item2; i++)  
            result[i] = random.NextDouble();  
        return random;  
    },  
    _ => {});
```

前面这个例子所用的 `Parallel.ForEach` 重载方法声明如下。

```
ForEach<TSource, TLocal>(  
    Partitioner.Create(0, numberOfSteps),  
    new ParallelOptions(),  
    () => { return new Random(MakeRandomSeed()); },  
    (range, loopState, random) =>  
    {  
        for (int i = range.Item1; i < range.Item2; i++)  
            result[i] = random.NextDouble();  
        return random;  
    },  
    _ => {});
```



```
OrderablePartitioner<TSource> source,
ParallelOptions parallelOptions,
Func<TLocal> localInit,
Func<TSource, ParallelLoopState, TLocal, TLocal> body,
Action<TLocal> localFinally)
```

Parallel.ForEach 循环为它的每一个操作任务创建一个 **Random** 类实例。这个实例将被作为参数传递给每个分区迭代。每个分区迭代负责返回下一个局部线程状态的值。在这个例子中，返回值始终是同一个传入对象。

Random 类的默认构造函数使用系统时钟来产生一个随机种子。如果构造函数被连续调用两次，默认的构造函数使用相同的种子是很有可能。因此，该代码可以保证，通过将不同的种子值作为一个参数传递给构造函数，对每一个新 **Random** 对象进行处理时，都使用一个不同的种子值。

这个例子是为了体现局部任务状态的应用。在 **Parallel** 类中，局部线程状态是由局部任务状态提供的。某一个任务保证在整个运行中只在一个线程上执行。

利用并行 **Random** 类来生成伪随机序列会导致不同线程中的重叠数字序列。如果应用程序确实需要统计学上很稳定的伪随机值，应该考虑使用 **RNGCryptoServiceProvider** 类或一个第三方库。欲了解更多有关生成随机数的信息，请阅读 2.8 节推荐的参考资料。

2.3.7 对并行循环使用自定义的任务调度程序

可以用自定义任务调度逻辑来代替使用线程池工作线程生成的默认任务调度程序。例如，可以使最大并行度对多个循环有效，而不只是针对默认的单循环。此外，如果想用一组专用线程来处理并行循环而不是从共享池中取出工作线程，那么，可以使用自定义任务调度程序。这里列举的仅仅是少数几个需要用到自定义任务调度程序的情况。

要指定一个自定义任务调度程序，就要设置一个 **ParallelOptions** 对象的 **TaskScheduler** 属性。示例如下。

连续两次调用默认的 **Random** 构造函数可能会用相同的随机种子。提供你自己的随机种子，以防产生重复的随机序列。

Random 类并不是适用于所有模拟场景的随机数生成器。此处使用它只是为了演示非线程安全的类。

如果内置的任务调度程序不能满足需求，可以用一个自定义任务调度程序。这是并行循环扩展机制中的一种。

```
int n = ...
TaskScheduler myScheduler = ...
var options = new ParallelOptions()
    { TaskScheduler = myScheduler };
Parallel.For(0, n, options, i =>
{
    // ...
});
```

以下是本代码示例中使用到的 `Parallel.For` 声明。

```
Parallel.For(int fromInclusive,
            int toExclusive,
            ParallelOptions parallelOptions,
            Action<int> body);
```

更多有关任务和任务调度程序的信息，请阅读 3.5 节。

PLINQ 查询不能指定一个自定义任务调度程序。

2.4 反模式

反模式是需要我们关注之处。它们突出了一些需要仔细考虑的问题以及问题域。

2.4.1 步长不为 1

如果需要使步长不为 1，可以改变循环体中的循环索引或用需要的值进行并行 `foreach` 操作。另外要知道，步长不为 1 意味着存在一个数据依赖。在将某个计算转换成并行循环前，请仔细分析是否合适。

在顺序程序中，除非步骤执行的先后顺序会有所影响，不然的话，很少按从高值到低值的顺序进行迭代。如果循环步长为负，循环的先后顺序

注意：
步长不为 1 表明
存在循环依赖。

可能会有所影响，且迭代之间可能相互不独立。

这里有一些有用的技巧。在将顺序循环转换成并行循环之前，可以暂时颠倒索引的顺序(例如，使它们从高到低)。如果顺序代码仍然正常运行，这就是一个证明循环步骤相互独立的证据。(此测试并非万无一失，你还需要了解算法。)

2.4.2 隐藏的循环体依赖

没有正确分析循环依赖是软件缺陷的常见原因。要注意，所有并行循环都不能包含隐藏的依赖。这是一个很容易犯的错误。

试图在并行迭代中共享如 `Random` 或 `DbConnection` 类的情况就是一个微妙依赖关系的例子，这些类是线程不安全的。

当循环体不是完全相互独立时，仍然有可能使用并行循环。然而，在这些情况下，必须确保所有的共享变量都是受保护和同步的，而且，必须理解任何你所添加的同步化的性能特点。添加同步化会大大降低并行程序的性能，但是，忘记添加必要的同步化会导致程序出现灾难性的故障且很难恢复。

如果循环体不是相互独立的，例如，用迭代去计算总数时，可能需要运用基于并行循环的变化形式，详情参见第 4 章。

2.4.3 少量迭代的小循环体

如果对那些只有有限个数据元素需要处理的非常小的循环体使用并行循环，可能不会获得性能上的改进。在这种情况下，并行循环本身所要求的开销将成为主要开销。仅仅将顺序的 `for` 循环改为 `Parallel.For`，不一定会产生好的结果。

2.4.4 处理器的超额申请和申请不足

随意增加并行度有可能会造成处理器的超额申请，当计算密集型的工

作线程数比内核数量多很多时,这种情况就会发生。测试表明,一般来说,并行任务工作线程的最佳数目等于用内核数除以每个任务所用内核的平均百分数。例如,对于四核且每个任务平均百分之五十的内核利用率的情况,为达到最大吞吐量,需要 8 个工作线程。当工作线程的数量增加到超过这个数时,额外的上下文切换将会导致额外的成本,而处理器的使用率没有任何改善。

另一方面,随意限制并行度会造成处理器的申请不足。任务太少就会错过有效利用可用处理器内核的机会。如果知道应用程序中还有其他的任务也在并行运行,可能需要限制并行度。

在大多数情况下,.NET Framework 中的内置负载均衡算法是进行权衡的最有效方法。它们协调并行循环和其他同时运行的任务之间的资源。

2.4.5 混合 Parallel 类和 PLINQ

PLINQ 查询由 `ParallelQuery<T>` 类实例表示。这个类实现了 `IEnumerable<T>` 接口,所以,可以将 PLINQ 查询当做 `Parallel.ForEach` 循环的源集合,但是,并不推荐这样做。与此相反,对于 `ParallelQuery<T>` 实例,推荐使用 PLINQ 的 `ForAll` 扩展方法。PLINQ 的 `ForAll` 扩展方法执行了与 `Parallel.ForEach` 类型相同的迭代,除此之外,它避免了不必要的合并和重新分区的步骤,否则将执行这些步骤。

如果需要遍历一个 PLINQ 结果,请使用 `ForAll`。不要在这种情况下使用 `ForEach`。

以下是一个有关如何使用 `ForAll` 扩展方法的例子。

```
var q = from d in data.AsParallel() ... select F(d);

q.ForAll(item =>
{
    // ... process item
});
```

2.4.6 输入枚举中的重复

如果使用的是 `Parallel.ForEach` 方法,在代码中,枚举中的重复对象

并行循环中不允许重复实例。如果一个对象在循环输入中出现了多次,有可能两个并行线程同时更新该对象。

表示的是一个不安全的竞争条件。如果一个对象(即,一个类的实例)在循环输入中出现了多次,有可能两个并行线程会同时更新该对象。

2.5 设计说明

以下是一些有关使用并行循环的细微之处。

2.5.1 自适应分区

Parallel 类适应从几个到几百万的各种迭代数量。

.NET 中,并行循环使用一个自适应分区技术,这里,操作单元的大小随时间推移增大。自适应分区既能满足小型输入范围的需求,也满足大型输入范围的需求。

2.5.2 自适应并发

在 .NET Framework 4 中,Parallel 类和 PLINQ 分别适用于稍有不同的线程模型。PLINQ 用固定数量的任务去执行一个查询;默认情况下,它创建与计算机中的逻辑内核相同数量的任务,或者,PLINQ 将使用传递给 WithDegreeOfParallelism 方法的值来决定任务数(如果该值被指定了的话)。

相反,默认情况下,Parallel.ForEach 和 Parallel.For 方法可以用一个变化的任务数。这就是为什么 ParallelOptions 类有一个 MaxDegreeOfParallelism 属性而不是一个 MinDegreeOfParallelism 属性。目的在于,系统可以使用比请求的线程数还少的线程来处理循环。

通过允许并行任务的工作线程数量随时间变化,.NET 线程池可以动态地适应不断变化的操作负载。运行时,系统观察线程数的增加是提高还是降低总吞吐量,并由此调整工作线程数量。

如果运行单步的并行循环花费了几秒或更久,那么,请提高警惕。这种情况会发生在接口密集型的操作负载以及冗长的计算中。如果需要很长

的循环时间，可能会遇到工作线程无限增加的情况，因为.NET ThreadPool类的线程注入逻辑使用试探法来防止线程的缺乏。当目前线程池的工作项运行了很长时间，这种试探法会稳步地增加工作线程数。这样做的动机是为了在线程池完全被堵塞的情况下增加更多的线程。遗憾的是，如果实际上操作正在执行，多线程不一定是你想要的。然而，.NET Framework无法区分这两种情况。

如果循环的每一步都需要很长时间，可能会看到比你计划的还多的线程数。

如果通过程序概要分析观察到由系统选择的线程数太大，可以用ParallelOptions类来限制线程数。也可以用ParallelOptions类的SetMaxThreads方法来限制线程数。通常，用ParallelOptions更好，因为SetMaxThreads有一个全局性的、进程级的影响，这种影响可能是你不想要的。

2.5.3 支持嵌套循环和服务端应用程序

.NET Framework 并行扩展允许嵌套并行循环。在这种情况下，运行环境会调整处理器资源的使用。你会发现，对于那些使用嵌套的并行循环，每个循环使用的线程数比非嵌套循环要少。

一个相关问题就是服务器应用程序中的并行循环处理。Parallel类试图用一种与处理嵌套循环完全相同的方式来处理服务器应用程序中的多个应用程序域。如果该服务器应用程序已使用所有可用的线程池线程来处理其他ASP.NET请求，那么，并行循环就只能在调用了它的线程上运行。如果操作负载减少，其他线程变为可用且没有其他ASP.NET操作要处理，那么，循环将开始使用额外的线程。

2.6 相关模式

并行循环模式是并行合并模式的基础，合并模式将是第4章的主题。

2.7 练习

1. 以下问题中有哪些可以用本章中所讲述的并行循环技术来解决？
 - a. 对一个有一百万个元素的内存数组进行排序。
 - b. 按字母表排列的顺序将每一行的单词从一个文本文件中读出。
 - c. 将一个集合中的所有数相加，以获得一个总和。
 - d. 从两个集合中成对添加数字，以获得一个总数的集合。
 - e. 统计一个文本文件集合中每一个单词出现的总次数。
 - f. 找出文本文件集合中每个文件中出现频率最高的单词。
2. 从练习 1 中选择一个合适的问题。用代码实现顺序循环、并行循环和 PLINQ 三种解决方法。
3. 在信贷审核例子中，账户是什么类型？`account.AllAccounts` 是什么类型？`account.AllAccounts.AsParallel()` 是什么类型？
4. 是否可用一个 PLINQ 查询作为 `Parallel.ForEach` 循环的源？是否推荐这样做？解释你的答案。
5. 在 CodePlex 网站(<http://parallelpatterns.codeplex.com>)上，做一个信贷审核示例代码的性能分析。使用命令行选项去单独改变迭代次数(账户号码)和在每个迭代体中执行的操作数(信用记录中的月数)。记录由三个版本的程序报告的执行时间，使用一些不同的月数和账号数的组合。在不同内核数和不同执行负载(从其他应用程序得出的)的计算机上重复上述测试。
6. 在 CodePlex 上修改信贷审核的例子，以便可以获得 `MaxDegreeOfParallelism` 属性。在有不同内核的计算机上运行时，观察这个属性的不同值对执行时间的影响。

2.8 扩展阅读

本书中的例子使用了最新的 C# 版本的功能和库。相关参考书籍有 Bishop、Albahari 等人的著作。这些书包含了有关 lambda 表达式的信息。由 Mattson 等人编写的书介绍了与特定语言和库无关的并行编程的软件设计模式。本章 2.3 节中的很多例子都是来自于 Toub 的报告，该报告包含了更多的范例和变化版本。要获得更多关于分区的信息，请参考 Toub 撰写的有关这个主题更深入的文章。

- Albahari J, Albahari B. *C# 4 in a Nutshell*. fourth edition. O'Reilly, 2010.
- Bishop J. *C# 3.0 Design Patterns*. O'Reilly, 2008.
- Mattson T G, Sanders B A, Massingill B L. *Patterns for Parallel Programming*. Addison-Wesley, 2004.
- Mattson T G. Use and Abuse of Random Numbers(video). 2008-02-14. <http://software.intel.com/en-us/videos/tim-mattson-use-and-abuse-of-random-numbers/>.
- Toub S. Patterns of Parallel Programming: Understanding and Applying Parallel Patterns with the .NET Framework 4 and C#. 2009. <http://www.microsoft.com/downloads/details.aspx?FamilyID=86b3d32b-ad26-4bb8-a3ae-c1637026c3ee&displaylang=en>.
- Toub S. Custom Parallel Partitioning With .NET 4. 2010-04-26. <http://www.drdoobbs.com/visualstudio/224600406>.



第 3 章 并行任务

第 2 章展示了如何使用并行循环将单个操作应用到多个数据元素上，即数据并行性。如果有一些能够同时运行的异步操作，情况又会怎样？本章将会对此做出解释。在这样的情况下，可以将一个程序的控制流分离成能够并行执行的任务，这就是任务并行性。并行任务模式有时也被称为分支/合并(Fork/Join)模式或者主/从(Master/Worker)模式。

并行任务是能够同时运行的异步操作。这种方法也被称为任务并行性。

数据并行性和任务并行性是一个系列的两个极端。数据并行性发生在将单个操作应用到多个输入的时候，而任务并行性涉及多种操作，每种操作都有各自的输入。

在微软的 .NET Framework 中，任务是通过 `System.Threading.Tasks` 命名空间中的 `Task` 类来实现的。它的静态属性 `Task.Factory` 是 `TaskFactory` 类的一个实例，它被用来创建和调度新任务。可以通过调用任务的 `Wait` 方法来等待它完成，也可以等待多个任务完成。如果从分支/合并的角度来考虑，那么 `StartNew` 方法就代表分支操作，`Wait` 方法就代表合并操作。

并行任务在 .NET 中是通过 `Task` 类来实现的。

调度是并行任务的一个重要方面。新任务并不一定会像线程那样直接开始运行，而是被放在一个工作队列里面。当任务调度程序将与它对应的任务从队列中取出时(一般是分配到 CPU 的时候)，任务才开始执行。任务调度程序会试着通过控制系统的并发度来优化整体吞吐量。只要有足够多的任务并且它们之间完全没有串行依赖，那么程序性能与可用内核数成比例，这样的任务就表现出第 1 章中介绍过的潜在并行性的特征。

基于任务的应用程序的另一个重要方面就是处理异常的方法。

.NET 中的任务会将异常搁置并在调用任务的一个 wait 方法时将它重新抛出。

在 .NET 中，任务在执行过程中抛出的一个未处理的异常稍后才会被观测到。例如，调用任务类的 wait 方法时，过一段时间后就会自动观测到被搁置的异常。这时，在 wait 方法的调用过程中该异常又被抛出了一次，这样，在并行程序中就能使用在顺序程序中使用的异常处理方法。

3.1 基础知识

每个任务都是一个顺序操作的过程，然而任务之间通常可以并行运行。在 .NET 中，任务也是一个拥有自己的属性和方法的对象。下面是一些顺序代码。

```
DoLeft();  
DoRight();
```

我们假设 DoLeft 和 DoRight 方法都是独立的，也就是说它们都不会在对方可能读数据的内存或文件上写数据。因为它们是独立的，所以可以使用 Parallel 类的 Invoke 方法来并行地调用它们。示例如下。

```
Parallel.Invoke(DoLeft, DoRight);
```

Parallel.Invoke 方法是并行任务模式下最简单的表达式。它为参数列表中的每个委托方法新建了并行任务。当所有的任务都完成时，Invoke 方法才会返回。

所有的并行任务并不一定会直接运行。根据当前的工作负荷和系统配置，任务可能会一个接一个地运行，也可能会同时运行。要想查看更多关于任务如何被调度的信息，请阅读 3.6 节。

Parallel.Invoke 中的委托方法可能会正常终止，也可能通过抛出异常来结束。在 Parallel.Invoke 执行的过程中抛出的任何异常都会被搁置，在所有任务完成时才被重新抛出。所有的异常都被作为一个 Aggregate

Parallel.Invoke 是 .NET 中创建一系列并行任务并等待它们全部完成的方法。

Exception 实例的内部异常重新抛出。欲查看更多信息和代码示例，请阅读 3.3.2 节。

从本质上讲，**Parallel.Invoke** 方法创建了新任务并等待它们完成。它使用 **Task** 类的方法来实现上述功能。下面是一个例子。

```
Task t1 = Task.Factory.StartNew(DoLeft);  
Task t2 = Task.Factory.StartNew(DoRight);  
  
Task.WaitAll(t1, t2);
```

TaskFactory 类的 **StartNew** 方法会创建并调度一个新任务，这个新任务执行的是作为参数提供的委托方法。可以通过调用 **Task.WaitAll** 方法来等待所有并行任务完成。（如果要等待单个任务，就使用 **Wait** 方法。）

使用 **StartNew** 方法新建任务时，这个新任务就被添加到一个工作队列中等待最终执行，但它不会直接开始运行，需要等到它的任务调度程序将它从工作队列中拿出来时才会开始，因此可能会直接开始，也可能在不久后开始。

目前你所看到的示例都很简单，但它们足以说明很多情况。欲查看更多使用任务的方法，请阅读 3.3 节。

3.2 示例

这个并行任务的例子是一个图像处理程序，其中的图像是由图层组成的。来源不同的图像被独立地进行处理，然后被一个叫做透明混合处理的过程组合起来。这个过程将半透明的图层叠加起来，从而组成一个图像。

这些要组合的源图像都是不同的，每个图像都被施加了不同的图像处理操作。也就是说，这些图像处理操作必须分别施加在每个源图像上，而且必须在这些图像被混合之前完成。在这个例子中，只有两个源图像，操

我们需要 **t1** 和 **t2** 两个任务在这个例子中演示正确的异常处理。如果没有第二个任务 **t2**，那代码便不能保证观测到任务 **t1** 执行过程中抛出的异常。要想查看更多信息，请阅读 3.3.2 节。

使用 **StartNew** 方法来创建一个任务并对它进行调度。然后使用 **Task.WaitAll** 方法来等待一个以上的任务完成。

作也很简单：改变灰度和旋转。在实际的例子中应该会有更多的源图像和更复杂的操作。

· 如下为顺序代码。这个例子的完整源代码在 <http://parallelpatterns.codeplex.com> 的 Chapter3\ImageBlender 文件夹中。

```
static int SequentialImageProcessing(Bitmap source1,
                                     Bitmap source2,
                                     Bitmap layer1,
                                     Bitmap layer2,
                                     Graphics blender)
{
    SetToGray(source1, layer1);
    Rotate(source2, layer2);
    Blend(layer1, layer2, blender);

    return source1.Width;
}
```

在这个例子中，source1 和 source2 是位图，它们是最初的源图像，layer1 和 layer2 也是位图，它们附带了混合图像所需的附加信息，blender 是一个 Graphics 实例，它负责进行混合操作并引用最终图像对应的位图。在程序内部，SetToGray、Rotate 还有 Blend 方法使用了 .NET 的 System.Drawing 命名空间下的方法来进行图像处理。最后一条语句返回一个调用者用来打印进度信息的整数。

SetToGray 方法和 Rotate 方法是完全相互独立的，也就是说可以在不同的任务中执行它们。如果有两个以上的内核是可用的，这两个任务可能会并行运行，而且对应的两个图像处理操作将在较短的时间内完成(相比于顺序运行的版本)。

并行代码明确地使用了任务。

```
static int ParallelTaskImageProcessing(Bitmap source1,
    Bitmap source2, Bitmap layer1, Bitmap layer2,
    Graphics blender)
{
    Task toGray = Task.Factory.StartNew(() =>
        SetToGray(source1, layer1));
    Task rotate = Task.Factory.StartNew(() =>
        Rotate(source2, layer2));
    Task.WaitAll(toGray, rotate);
    Blend(layer1, layer2, blender);
    return source1.Width;
}
```

这段代码调用 **Task.Factory.StartNew** 方法来创建并运行两个任务，它们分别执行 **SetToGray** 和 **Rotate** 方法，然后调用 **Task.WaitAll** 方法在混合图像之前等待两个任务完成。

也可以使用 **Parallel.Invoke** 方法来实现并行。**Parallel.Invoke** 的语法非常方便。下面的代码显示了这一点。

```
static int ParallelInvokeImageProcessing(Bitmap source1,
    Bitmap source2, Bitmap layer1, Bitmap layer2,
    Graphics blender)
{
    Parallel.Invoke(
        () => SetToGray(source1, layer1),
        () => Rotate(source2, layer2));
    Blend(layer1, layer2, blender);
    return source1.Width;
}
```

在这里，**Parallel.Invoke** 方法的参数隐式识别了这两个任务。这次调用将在所有任务完成时返回。

3.3 变化形式

本节将描述.NET 并行任务模式的实现中的变化形式。

3.3.1 取消任务

.NET 中的任务使用一个合作取消模型。

在.NET 中，一个取消请求并不能强制任务结束，任务会使用一个合作取消模型。也就是说一个正在运行的任务必须在合适的时间段轮询取消请求，然后通过将自己回调入库来停止运行。

.NET Framework 使用了两种不同的类型。一个让程序能够请求取消，另一个可以检查是否存在取消请求。`CancellationTokenSource` 类的实例被用来请求取消，而 `CancellationToken` 类的实例会指出是否存在取消请求。示例如下。

```
CancellationTokenSource cts = new CancellationTokenSource();
CancellationToken token = cts.Token;

Task myTask = Task.Factory.StartNew(() =>
{
    for (...)
    {
        token.ThrowIfCancellationRequested();

        // Body of for loop.
    }
}, token);

// ... elsewhere ...
cts.Cancel();
```

`CancellationTokenSource` 对象中包含了一个取消标记。有两个地方会用到这

个标记,它是 `StartNew` 方法的一个参数,调用它的 `ThrowIfCancellationRequested` 方法可以检测并处理一个取消请求。虽然这个例子没有显示这一点,但可以查看这个取消标记的 `IsCancellationRequested` 属性来查看是否存在取消请求。例如,如果有个任务正在被取消的过程中,而你要对它进行一些清理工作时,大概就会这么做。

如果这个取消标记表明存在取消请求,那么 `ThrowIfCancellationRequested` 方法就会创建一个 `OperationCanceledException` 实例并传入取消标记,然后它便抛出这个异常。这个异常是通知 .NET Framework 这个任务已被取消的信号,因此, `OperationCanceledException` 是不能被任务中的用户代码处理的(通常是被取消的任务之外的代码)。如果跟着这节描述的步骤来操作,那么任务停止时它的状态属性将被设置成枚举值 `TaskStatus.Canceled`。

为保证一个正在运行的任务过渡到 `Canceled` 状态, `StartNew` 方法必须要有一个取消标记来作为它的参数,同时任务必须抛出一个包含相同取消标记的 `OperationCanceledException` 异常。

必须满足几项要求才能保证一个任务过渡到 `Canceled` 状态。

在一个循环中反复检查是否存在取消请求(每次循环都执行一小部分任务)会对应用程序性能产生消极影响,但不这样做,可能导致应用程序响应取消请求的时间过长。例如,在一个基于 GUI 的交互式应用程序中,每秒钟最好检查取消请求一次以上;运行在后台的应用程序轮询取消请求的频率可以比较小,大概每两秒到十秒检查一次。概要分析应用程序之后就可以得到它的性能数据,接着就能通过这些数据来决定在代码中检测取消请求的最佳位置。你需要知道代码中的哪些位置能够让你每隔相等的时间轮询一次取消请求,而概要分析能够帮你做到这点。

一个任务能不能过渡到取消状态是很重要的。如果做法不正确,那么当非正常取消的任务进行垃圾回收时,进程就有可能异常终止。一种典型的失误就是忘记将取消标记传给 `StartNew` 方法。另一种典型失误就是忘记抛出包含必需的取消标记的 `OperationCanceledException` 异常。

如果某个取消标记被传到 `StartNew` 方法中,而且这个标记在新任务还没开始执行之前就已经检测到取消请求,那么新任务的状态就会过渡到 `Canceled` 状态而不会运行。

欲查看关于取消请求的若干表现细节,请阅读 3.5.2 节和 3.5.4 节。

3.3.2 处理异常

如果任务对象的委托方法在执行时抛出一个未处理的异常, 这个任务就会终止, 而且它的状态属性会被设置成枚举值 `TaskStatus.Faulted`。这个未处理的异常暂时未观测到, 任务并行库(TPL)会捕获这个异常并将它的细节记录在与这个任务对象相关联的内部数据结构中。捕获被搁置的异常并把它嵌入一个新的异常中的过程就被称作“观测未处理的异常”。在很多情况下, 未处理的异常会在另外一个线程中被观测到, 而不是执行这个任务的线程。

1. 观测未处理的异常的方法

使用下面几种方法可以观测未处理的异常。

调用出错了的任务的 `Wait` 方法可以观测到这个任务的未处理异常。这个异常也在 `Wait` 方法的上下文中被抛出。`Task` 类的静态 `WaitAll` 方法可以用于在一次方法调用中观测到多个任务的未处理异常。`Parallel.Invoke` 方法在内部调用了 `WaitAll`。所有任务的异常在一个 `AggregateException` 对象中集合并在 `WaitAll` 或 `Wait` 方法的上下文中抛出。

查看出错了的任务的 `Exception` 属性可以观测到这个任务的未处理异常。此属性返回汇总异常对象, 查看这个值并不会自动让异常抛出; 查看 `Exception` 属性的值时, 该异常只是被认为被观测到了。如果要检查未处理异常但并不想抛出它, 就使用 `Exception` 属性而不是 `Wait` 或 `WaitAll` 方法。

如果一个出错了的任务的未处理异常在任务对象进行垃圾回收时还没有被观测到, 系统就会进行特殊的处理。欲查看更多信息, 请阅读 3.5.4 节。

一定要注意观测每个任务的未处理异常。如果不这样做, .NET 的异常调整策略会在任务进行垃圾回收时终止进程。

2. 汇总异常

在某些情况下，例如在等待多个任务时，可能需要观测多个未处理的异常。为此，运行时(也就是 runtime，下同)会收集这些异常并将它们包装在一个 `AggregateException` 实例中。TPL 在观测者的上下文中(也就是在调用 `Wait` 方法的线程中)抛出这个汇总异常。可以捕获这个汇总异常并检查它的 `InnerExceptions` 属性，从而对原来的异常分别进行处理。即使只有一个任务的异常被观测到，它仍然会被包装到一个 `AggregateException` 中。抛出的汇总异常也保存了内部异常的堆栈跟踪(stack trace)信息，如果异常在当前上下文中再次抛出，这些信息就可能会被覆盖。

任务并行库使用 `AggregateException` 对象来传递异常。

跟所有在任务内部出现的异常一样，取消一个任务时生成的 `OperationCanceledException` 实例会成为一个 `AggregateException` 的内部异常，该任务会在调用它的 `Wait` 方法的线程上下文中抛出这个汇总异常。然而，如果遵循本章前面 3.3.1 节中描述的取消规则，这个任务的最终状态就会是 `TaskStatus.Canceled` 而不是 `TaskStatus.Faulted`。

3. Handle 方法

`AggregateException` 类有几个辅助方法简化了内部异常的处理。它的 `Handle` 方法为汇总异常中的每个内部异常调用一个用户提供的委托方法。这个委托应该在处理了内部异常时返回 `true`，没有处理内部异常时返回 `false`。示例如下。

使用 `Handle` 方法来循环访问内部异常。

```
try
{
    Task t = Task.Factory.StartNew( ... );
    // ...
    t.Wait();
}
catch (AggregateException ae)
```

```
{
    ae.Handle(e =>
    {
        if (e is MyException)
        {
            // ... handle exception ...
            return true;
        }
        else
        {
            return false;
        }
    });
}
```

当 **Handle** 方法为每个内部异常调用用户提供的委托时，它记录了调用的结果。处理完所有的内部异常后，它会检查有没有一个或多个内部异常的结果是 **false**，那样的异常就是没有被处理的。如果有未处理的异常，**Handle** 方法会创建并抛出一个新的汇总异常，里面的内部异常也就是这些未被处理的。

如果 **Handle** 方法不能满足要求，可以自己写代码来循环访问 **AggregateException** 对象的 **InnerExceptions** 属性。

4. Flatten 方法

AggregateException 类的 **Flatten** 方法适用于任务被嵌套在其他任务中的情况。在这种情况下，汇总异常有可能会将其他汇总异常作为它的内部异常。**Flatten** 方法会产生一个新的异常对象，该对象将所有被嵌套的汇总异常的内部异常并入顶层汇总异常的内部异常中。换句话说，它将一个由内部异常组成的树转换成一个简单的序列。一般把 **Flatten** 和 **Handle** 方法放在一起使用，示例如下。

Flatten 方法将一个由异常组成的树转换成一个序列。

```
try
{
    Task t1 = Task.Factory.StartNew(() =>
    {
        Task t2 = Task.Factory.StartNew(() =>
        {
            // ...
            throw new MyException();
        });
        // ...
        t2.Wait();
    });
    // ...
    t1.Wait();
}
catch (AggregateException ae)
{
    ae.Flatten().Handle(e =>
    {
        if (e is MyException)
        {
            // ... handle exception ...
            return true;
        }
        else
        {
            return false;
        }
    });
}
```

使用 `Task.WaitAny` 方法来等待一组任务中的一个或多个任务完成。

3.3.3 等待第一个任务完成

`Task` 类有一个叫做 `Task.WaitAll` 的方法可以用于等待一组任务结束。然而也可以通过调用 `Task.WaitAny` 方法来等待第一个任务完成。

下面是一个例子。

```
var taskIndex = -1;

Task[] tasks = new Task[]
{
    Task.Factory.StartNew(DoLeft),
    Task.Factory.StartNew(DoRight),
    Task.Factory.StartNew(DoCenter)
};

Task[] allTasks = tasks;

// 逐个输出任务完成的通知
while (tasks.Length > 0)
{
    taskIndex = Task.WaitAny(tasks);
    Console.WriteLine("Finished task {0}.", taskIndex + 1);
    tasks = tasks.Where((t) => t != tasks[taskIndex]).ToArray();
}

// 观测可能发生的任何异常
try
{
    Task.WaitAll(allTasks);
}
catch (AggregateException ae)
{
    ...
}
```

这个例子演示了如何使用 `WaitAny` 方法在一组任务中的第一个任务完成时收到通知。代码会在每个任务完成时在控制台输出窗口中打印一行。`while` 循环在所有任务完成时退出。

你需要观测可能发生的任何异常，但要注意的是，异常不会被 `WaitAny` 方法观测到，所以要在使用 `WaitAny` 方法时添加一个检测异常的步骤。虽然这个例子使用 `WaitAll` 方法来观测异常，但也可以使用 `Exception` 属性或者 `Wait` 方法来达到这个目的。`Task` 类的 `IsFaulted` 属性可以用来检测任务内部是否发生未处理的异常。

异常从来不会被 `WaitAny` 方法观测到，但必须使用 3.3.2 节中描述过的技术保证所有的异常最终都被观测到。

3.3.4 推测执行

推测执行是另一种变化形式，带着对某种特定结果的预期执行操作就叫推测执行。例如，你可能预测当前运行的程序的输出值是 42，然后你启动依赖这个程序结果的下一个计算，并将 42 作为它的输入。如果第一个计算的结果是 42，那么也就成功地提前启动了有依赖的操作，从而实现了并行。如果第一个计算结果不是 42，就要使用正确的值重启第二个操作。

另一个推测执行例子发生在并行执行多个异步操作但只需要其中一个操作结果时。例如说，想使用三个不同的搜索任务来搜索一个东西，等最快的任务找到它之后，就不用等待另外两个搜索任务了，也就是等第一个任务完成后就取消剩下的任务。但你还是要观测所有任务中可能发生的异常。

下面是一个例子。

```
SpeculativeInvoke(SearchLeft, SearchRight, SearchCenter);
```

这个例子并行执行三个委托方法，这三个委托中只要有一个完成了，这个操作就成功了，其他没有完成的任务就要被取消。`SpeculativeInvoke` 方法会执行 `params` 参数数组中的每个委托。下面的代码显示了它的实现过程。


```
public static void SpeculativeInvoke(  
    params Action<CancellationToken>[] actions)  
{  
    var cts = new CancellationTokenSource();  
    var token = cts.Token;  
    var tasks =  
        (from a in actions  
         select Task.Factory.StartNew(() => a(token), token))  
        .ToArray();  
  
    // 等待最快的任务完成  
    Task.WaitAny(tasks);  
  
    // 取消所有其他任务  
    cts.Cancel();  
  
    // 等待取消请求并观测异常  
    try  
    {  
        Task.WaitAll(tasks);  
    }  
    catch (AggregateException ae)  
    {  
        // 筛选掉由于取消请求引起的异常  
        ae.Flatten().Handle(e => e is OperationCanceledException);  
    }  
    finally  
    {  
        if (cts != null) cts.Dispose();  
    }  
}
```

`WaitAny` 方法不会观测未处理的任务异常。这是因为你需要它返回的索引来告诉你哪个任务完成了(可能是因为抛出了异常才完成的)。因此,这个例子在取消了没有第一个完成的任务之后还调用了 `WaitAll` 方法,这样就可以在当前线程中观测到未处理的任务异常。上面的代码捕获 `WaitAll` 抛出的汇总异常并排除了 `OperationCanceledException` 实例。如果没有其他的未处理异常,代码就会正常运行下去,否则,其他未处理异常就要被作为一个新的汇总异常的内部异常重新抛出。

3.3.5 使用自定义的调度方式创建任务

可以自定义调度 .NET 任务的方式,方法是覆盖任务的 `Factory` 成员的方法使用的默认任务调度程序。例如,可以将一个自定义的任务调度程序作为参数提供给 `Task.Factory.StartNew` 方法的一个重载版本。

可以自定义调度任务的方式。

在某些情况下可能需要覆盖默认的调度程序。最常见的情况就是让任务运行在一个特殊的线程上下文时。可能是你使用库(例如 `Windows Presentation Foundation(WPF)`)提供的对象时,这些对象采用了线程亲和约束。其他情况也就是当默认任务调度程序的负载均衡试探法不适用于你的应用时。欲查看更多信息,请阅读 3.6.7 节。

如果没有指定专门的任务调度程序,那么所有新任务都会使用当前的任务调度程序,这是由静态属性 `TaskScheduler.Current` 的值来确定的。换句话说,子任务继承了父任务上下文中的任务调度程序。除非进行了特殊的指定,不然新的顶层任务(连同它们的子任务)都会使用默认的任务调度程序,这是由静态属性 `TaskScheduler.Default` 的值来确定的。默认的任务调度程序高度集成了 .NET 的线程池并处理各种各样的运行条件,对大部分应用都很适用。更多详情可参见 3.6 节。`TaskScheduler` 类的 `FromCurrentSynchronizationContext` 静态方法返回一个任务调度程序对象,该对象可以让任务运行在当前线程的 `SynchronizationContext` 属性指

定的任何地方，有时候就是在当前线程中，但并不一定。这样的任务调度程序对线程亲和很管用。并不是每个线程都有一个当前同步上下文，而且也没有 API 能够获取一个运行在非当前线程的同步上下文中的任务调度程序。例如，如果要调度一个 Windows 服务的主线程中的任务，就不能使用 `FromCurrentSynchronizationContext` 方法。

可以实现自己的任务调度程序类。欲查看更多信息，请阅读 3.5.3 节。

3.4 反模式

下面是一些使用任务时要特别注意的内容。

3.4.1 闭包捕获的变量

在 C# 中，闭包可以用 `args=>body` 形式(代表一个未命名的委托)的 `lambda` 表达式来创建。闭包的一个特征就是它们可能会引用它们的词法作用域之外定义的变量，例如在包含这个闭包的作用域内声明的局部变量。

C# 闭包的语义可能对某些程序员来说不很直观，而且容易犯错。如果不能理解它的语义，那你可能会发现被捕获的变量并不像你预料的那样，特别是在并行程序里。

在闭包中捕获一个循环索引变量一般都是错误的。注意这种常见的错误。

如果不考虑变量的作用域就直接引用它就会出现問題，示例如下。

```
for (int i = 0; i < 4; i++)
{
    // WARNING: BUGGY CODE, i has unexpected value
    Task.Factory.StartNew(() => Console.WriteLine(i));
}
```

你可能认为这段示例代码会按未知的顺序输出数字 1, 2, 3, 4, 但实际上它会输出其他值序列, 这取决于这些线程是如何运行的, 例如可能看到 4, 4, 4, 4。因为变量 `i` 是被 `for` 循环创建的所有闭包共享的。任务启动时, 这个被共享的变量 `i` 的值和任务被创建时 `i` 的值可能是不一样的。所有的任务共享了同一个变量。

解决办法就是在合适的作用域添加一个临时变量。

```
for (int i = 0; i < 4; i++)
{
    var tmp = i;
    Task.Factory.StartNew(() => Console.WriteLine(tmp));
}
```

这样就会按未知顺序输出 1, 2, 3, 4, 每个数字都会被输出。因为变量 `tmp` 是在 `for` 循环体的作用域之内声明的, 这样一来 `for` 循环的每次迭代都会生成一个 `tmp` 变量实例。(相比之下, `for` 循环的每次迭代都共享了变量 `i` 的唯一一个实例。)

这个漏洞是你为什么应该使用 `Parallel.For` 而不是自己编写循环的原因之一, 也是未接触过任务的程序员犯下的最常见的错误之一。

3.4.2 清理任务所需要的资源

新建一个任务时, 不能调用执行任务所需的对象的 `Dispose` 方法。如果 C# 的 `using` 关键字用得不够细心, 就很容易犯这个错误。示例如下。

```
Task<string> t;
using (var file = new StringReader("text"))
{
    t = Task<string>.Factory.StartNew(() => file.ReadLine());
}
// WARNING: BUGGY CODE, file has been disposed
Console.WriteLine(t.Result);
```

注意, 不要清理一个挂起的任务所需的资源。

这里的 `using` 关键字给代码添加了一个隐含的 `try/finally` 块，它会在变量离开它的作用域时调用它的值的 `Dispose` 方法。`using` 关键字不能用在被捕获的变量上。然而，如果知道哪个对象不再被需要，就应该调用它的 `Dispose` 方法。在这个例子中，`Dispose` 方法只能在读取了任务的 `Result` 属性后才能被调用。

3.4.3 避免撤销线程

使用 `Thread.Abort` 方法来终止任务会让 `AppDomain` 变成一种潜在的不可用状态，而且撤销一个线程池工作线程是从来不被提倡的。要取消一个任务，最好使用 3.3.1 节中描述的方法，而不是撤销任务的线程。

永远不要通过调用正在执行任务的线程的 `Abort` 方法来取消一个任务。

3.5 设计说明

这一节描述的是一些创建任务并行库时要注意的设计要素。

3.5.1 任务和线程

任务开始运行时，一个合适的任务调度程序会在它选择的线程内调用这个任务的用户委托。

任务在运行期间不会在线程之间移动。这是个有用的保障，因为它能让你放心使用线程亲和的抽象概念(例如微软的 Windows 操作系统的临界区域)，例如，`Monitor.Enter` 函数和 `Monitor.Exit` 函数在不同的线程中执行。

3.5.2 任务生命周期

`Task` 实例的 `Status` 属性跟踪了任务的生命周期。图 3.1 显示了本章描述的任务的生命周期。

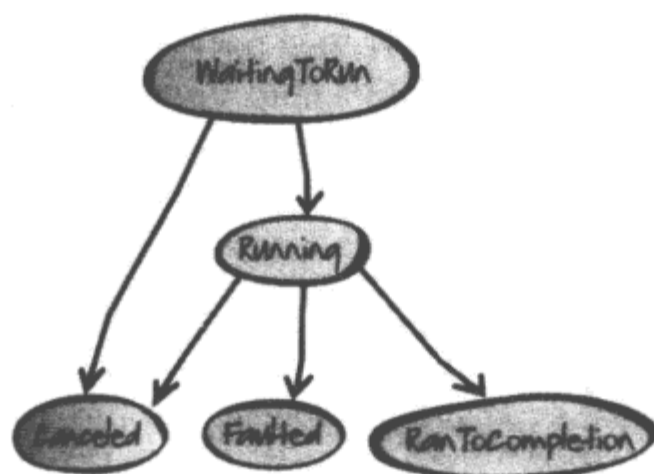


图 3.1 任务的生命周期

`TaskFactory.StartNew` 方法创建并调度一个新任务，这样任务的状态就变成 `TaskStatus.WaitingToRun`。负责管理这个任务的 `TaskScheduler` 实例最终在它选择的线程中执行任务的用户委托时，任务的状态就过渡到 `TaskStatus.Running`。任务开始运行之后，它会有三种可能的结果，如果任务的用户委托正常退出，那么任务的状态就过渡到 `TaskStatus.RanToCompletion`；如果任务的用户委托抛出一个未处理的异常，那么任务的状态就变成 `Task.Faulted`。

任务结束时的状态也可能是 `TaskStatus.Canceled`。要出现这样的情况，必须将一个 `CancellationToken` 作为参数传给 `Factory` 成员创建任务的方法。如果这个标记在任务开始执行之前就发出了一个取消请求，那么这个任务就不会被允许运行，而且这个任务的 `Status` 属性会直接过渡到 `TaskStatus.Canceled`，它的用户委托也不会被调用。如果这个标记在任务开始执行之后发出了一个取消请求，而且用户委托抛出了一个 `OperationCanceledException` 异常并且该异常的 `CancellationToken` 属性包含了任务创建时给出的标记，那么任务的 `Status` 属性就会过渡到 `TaskStatus.Canceled`。

在本书后面，会看到任务生命周期的两个变体。第5章介绍了生命周期也取决于先行任务的延续任务。第6章描述了生命周期取决于子任务的

任务，这些子任务在创建时使用了 `AttachedToParent` 创建选项。

还有另外一种任务状态叫做 `TaskStatus.Created`。这是在 `Task` 类的构造方法创建任务之后会立刻显示的状态；但还是建议你使用 `Factory` 成员的方法来创建任务，不要使用 `new` 操作符。

3.5.3 编写自定义的任务调度程序

.NET Framework 实现了两个任务调度程序：默认任务调度程序和将任务运行在目标同步上下文上的任务调度程序。如果这些任务调度程序不能满足你的应用需要，实现一个自定义的任务调度程序也是可以的。

有一些与创建自定义任务调度程序有关的高级方案。例如，如果通过多循环和多任务(而不是单个循环)来实现最大的并行度，就可以实现一个自定义任务调度程序。如果要实现可选择的调度算法，也可以编写一个调度程序，例如为了保证批任务处理过程中的公平性。如果要使用一个特定的线程集，例如 `Single Thread Apartment(STA)`线程，而不是线程池工作线程，同样可以实现一个自定义任务调度程序。

微软的并行示例代码包里面有个 `ParallelExtensionsExtras` 项目，可以参考它来创建自定义的任务调度程序。欲获取这些示例代码的地址，请查看 3.8 节。这个包实现了一个 `QueuedTaskScheduler` 类，它提供了多重全局任务队列，采用循环调度算法。自定义的调度程序不需要使用它自己的线程；它能限制当前允许运行的任务数量并仍然使用线程池。这是 `ParallelExtensionsExtras` 项目中的 `LimitedConcurrencyLevelTaskScheduler` 类采用的方法。

3.5.4 未观测到的任务异常

如果不让出错的任务抛出它的异常(例如调用 `Wait` 方法)，运行时就会根据当前的 .NET 异常机制在任务进行垃圾清理的时候，升级这个任务的未观测异常。未观测到的任务异常最终将被清理器线程上下文观测到。

清理器线程是调用即将进行垃圾清理的对象的 `Finalize` 方法的系统线程。如果在一个 `Finalize` 方法的执行过程中抛出了一个未处理的异常，那么运行时就会默认地终止当前进程，不会执行任何 `try/finally` 块或其他的清理事器，包括释放非托管资源句柄的清理事器。为阻止这种情况的发生，要非常小心，不要让应用遗漏任何未观测到的任务异常。也可以选择接收各种未观测到的任务异常的通知，只需预订 `TaskScheduler` 类的 `UnobservedTaskException` 事件并选择在异常抛到清理器上下文的时候对它们进行处理。

上面的最后一个方法在某些方案中可能会很有用，例如承载不可信并带有很难观测到的良性异常的插件程序。欲查看更多信息，请阅读 3.8 节推荐的参考资料。

在清理阶段，`status` 属性为 `Faulted` 的任务与 `Canceled` 状态的任务得到了不同的处理。任务的状态决定了由取消任务操作引起的未观测任务异常得到何种处理。如果被作为参数传给 `StartNew` 方法的取消标记和嵌入未观测的 `OperationCanceledException` 异常实例中的标记是同一个，这个任务就不会将操作取消异常传给 `UnobservedTaskException` 事件或清理器线程上下文。换句话说，如果遵循本章中描述过的执行取消的协议，那么未观测到的取消异常就不会被升级并传到清理器的线程上下文。

3.5.5 数据并行性和任务并行性之间的关系

`Parallel.Invoke` 方法被用来给它 `params` 数组中的每个委托创建任务。作为一个构思模型，这很好。但在实践中，它并不总是这样执行的。例如，如果在 `params` 列表中有很多委托，TPL 可能会顾及性能而使用一个并行循环来调用这些委托。

这也就突出了任务并行性和数据并行性之间的关系。如果每个操作对应的是一个委托，就可以使用数据并行性在每个数据(委托)上执行相同的操作(调用委托)。反过来，如果是数据并行问题，也可以使用任务并行方式来处理，也就是为每个操作启动一个任务。

3.6 默认任务调度程序

这一节描述的技术内幕适用于 .NET Framework 4。我们不能保证运行时或框架的未来发布版依旧这样运行。

所有的并行库都依赖于一个调度程序来组织并整理任务和线程的执行，TPL 也是这样。这一节我们将看到 .NET Framework 4 实现任务调度的内幕，这里提供的材料能帮你了解在使用 TPL 和 PLINQ 时会看到的运行特性。要注意这里描述的调度算法只是代表一种实现方法，这些算法并不是 TPL 本身强加的限制，未来版本的 .NET 可能会对任务的执行进行优化。

在 .NET Framework 4 中，默认调度程序是与线程池高度集成在一起的。如果使用默认任务调度程序，那么执行并行任务的工作线程便由 .NET 的 `ThreadPool` 类管理。一般来说，计算机上有多少内核，就至少有多少个工作线程。当任务数比可用工作线程数多时，一些任务就要排队等候，直到线程池提供一个可用的工作线程为止。

这种机制的一个例子就是线程池的 `QueueUserWorkItem` 方法。实际上，可以把默认任务调度程序看成一个改进的线程池，这种线程池中的工作项目会返回一个句柄，线程可以把它用作一个等待条件，而其中的未处理任务异常都会被转发到该等待上下文。指向工作项目的句柄也就是任务，调用任务的 `Wait` 方法时也就触发了等待条件。除此之外，随着内核数量的增长，默认线程调度程序的性能比线程池更好。下面看看它是如何运行的。

3.6.1 线程池

简单来说，线程池是由一个包含挂起工作项目的全局队列和一组运行工作项目的线程组成，一般采用先进先出(FIFO)的调度原则，如图 3.2 所示。

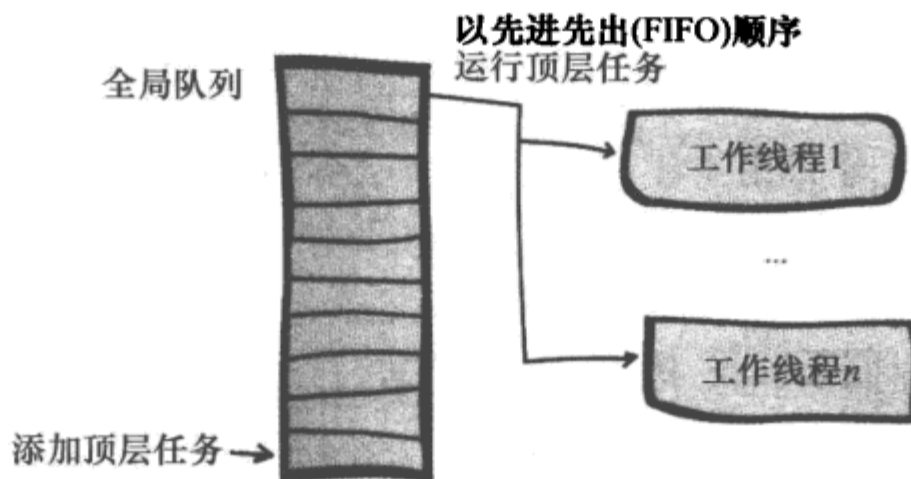


图 3.2 线程池

线程池难以处理有很多内核的情况，主要是因为线程池只有一个全局工作队列，这个全局队列的两端一次都只能让一个线程访问，这就可能会成为一个瓶颈。如果只有少数笨重的工作项目和有限数量的内核，那么全局队列的同步管理开销(也就是保证一次只有一个线程访问的花费)是比较小的。例如，当内核数量小于等于 4，或者每个工作项目占用大量处理器周期时，这种花费是可以忽略的。然而，随着内核数量的增长和每个工作项目工作量的减少(由于实现轻量任务的并行性需要利用更多的内核)，传统线程池的同步化的开销开始变得不容忽视。

同步是一个涵盖性术语，包括了很多协调多线程应用的活动的技术。锁是同步技术的一个熟悉的概念，线程使用锁来保证多个线程不会在相同的时间修改内存的同一个位置。从本质而言，同步的所有形式都是阻塞一个线程(也就是不运行)直到满足一个条件。

.NET 中的任务能运行在有很多内核的情况下，这些任务也可以是足够轻量的，可以执行很少量工作，大概是几百或几千个 CPU 周期。在.NET 中，让一个拥有成千上万任务的应用程序高效运行是可能的。要处理这种规模，我们需要一个更加分散的管理方法来调度，而不是只使用一个全局队列。

3.6.2 分散管理的调度技术

.NET Framework 为线程池中的每个工作线程提供了一个局部任务队列。局部任务队列分散了队列管理的压力并避免了全局队列中的工作项目的大量顺序访问。这样让应用程序的不同部分具有自己的工作队列也就避免了主要的性能瓶颈问题，如图 3.3 所示。

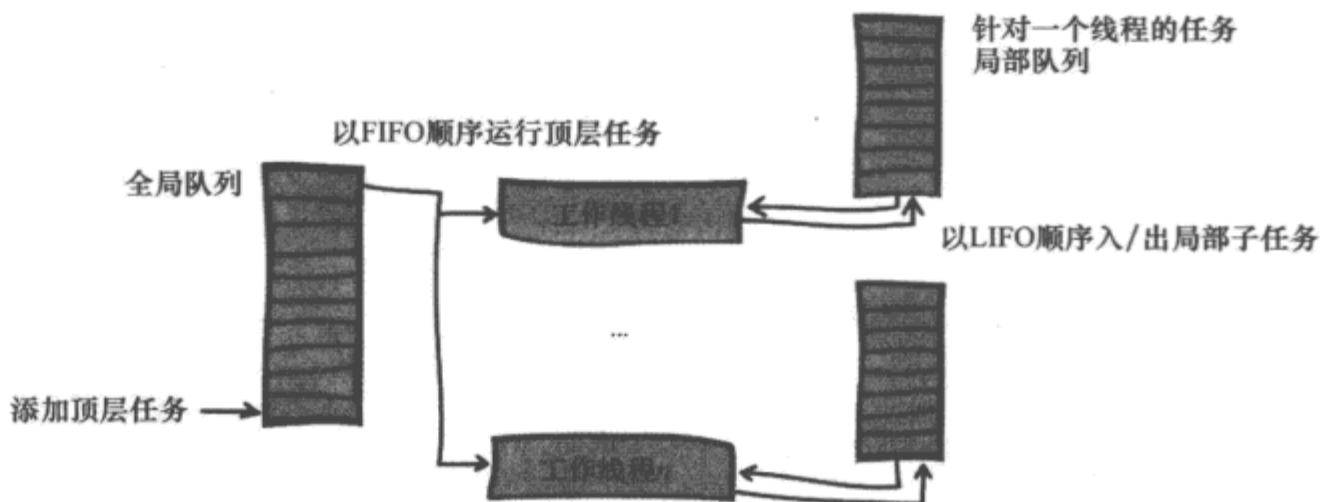


图 3.3 局部任务队列

可以看到，图中的任务队列的个数和工作线程的个数是相等的，另外还有一个全局队列，所有这些队列会同时工作。其工作的基本理念如下，当需要增加一个新任务时，它有时会被添加到一个线程的局部队列中，而不是全局队列；当线程准备让一个新任务运行时，它有时可以在自己的局部队列中找一个，而不必到全局队列中去找。当然，如果任务所在的线程不是线程池中的工作线程，它还是要待在全局队列里的，比起添加到局部队列，这样做总会导致较大的同步开销。

一般情况下，访问局部队列只需很小的同步开销，工作项目在局部添加和移除都很快。能有这样的效率是因为实现局部队列的时候使用了一种特殊的并行数据结构，叫做 **work-stealing** 队列。**work-stealing** 队列是一种拥有一个私有端和一个公有端的双端队列。这种队列允许在私有端进行无锁的入队和出队，但在公有端的操作则需要较大的同步开销。当队列的长

度较小时，由于实现所采用的锁定策略，队列两端都需要同步。

采用分散的调度方法后，任务执行顺序就没有原来只有一个全局队列时那么容易预测了。虽然全局队列采用 FIFO 执行顺序，但局部 work-stealing 队列使用 LIFO 顺序来避免同步开销。这样整体吞吐量可能会好一些，因为线程只会在用完自己的局部队列中的任务之后才会去访问同步开销较大的全局队列。

3.6.3 work stealing 策略

当线程的局部工作队列和全局队列都空了的时候会怎么样呢？其他工作线程的局部队列里可能会有一些任务。这时我们就可以用到 work stealing 策略，如图 3.4 所示。

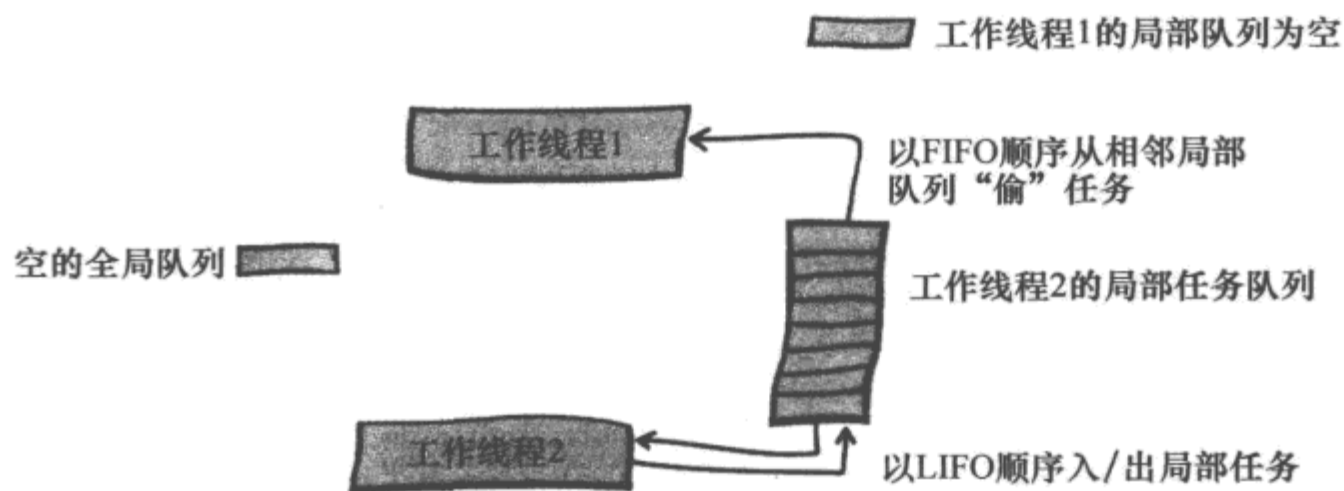


图 3.4 work stealing 策略

图 3.4 显示的是当一个线程的局部队列和全局队列中都没有条目的时候，系统就会从另外一个工作线程的局部队列中“偷”任务。为了最小化同步开销，系统会从第二个线程的 work-stealing 队列的公共端获取任务。这也就是说，除非队列非常短，否则第二个线程仍然能继续在自己的局部队列的私有端进行入队和出队操作(以最小的同步开销)。

同时采用 LIFO 和 FIFO 执行顺序还有其他有意思的好处，这些好处源于一般的应用程序的工作分配模式。实践表明 LIFO 顺序对局部队列来

说比较有意义，因为它减小了高速缓存缺失率。刚刚被放到工作队列里的那些东西很可能会引用仍然停留在系统内存缓存区里的对象。要好好利用高速缓存，就要优先处理最近添加的任务。

许多并行算法都采用了一个跟递归类似的分治法，正如第 6 章所述，较大的子任务将优先入队。采用 FIFO 执行顺序，这些大家伙是第一个被其他线程处理的。将一个较大的任务传递给另一个线程后，该线程在短期之内便不需要再偷其他任务。这种大任务执行的时候，它的子任务会在新的线程的局部队列中入队和出队，这种调度大任务的方法的效率是非常高的。

3.6.4 全局队列中的顶层任务

`Task` 类的 `Factory` 成员的方法如果是被非线程池工作线程调用的，那么任务就要被放在全局队列里。（当然，前提是 `Factory` 成员的方法必须允许使用默认任务调度程序。本节中的信息只适用于被默认任务调度程序管理的任务。）

你也可以强制默认任务调度程序将任务放在全局队列里，只需将这个任务的创建选项 `PreferFairness` 传递给 `Factory` 成员的方法即可。

在本书中，全局队列中的任务都被称作顶层任务，顶层任务的行为特征与线程池的 `QueueUserWorkItem` 方法创建的工作项目的行为特征大致是一样的。

3.6.5 局部队列中的子任务

当线程池工作线程调用 `Task` 类 `Factory` 成员的方法时，默认任务调度程序会将新任务放在这个线程的局部任务队列里，这比放到全局队列中去要快一些。

默认任务调度程序假设最小化子任务最坏情况下的延迟是不重要的，而它的目的是优化系统的总体吞吐量。如果你认为每次从另一个任务或一

个线程池工作项目中创建一个任务时，你执行的操作是某个更大的操作（例如某个顶层任务）的一部分的话，这就是有意义的。在这样的情况下，唯一有影响的延迟就是顶层任务的延迟。因此，默认任务调度程序不会关心子任务的 FIFO 顺序。虽然这种假设并不在所有情况下成立，但理解好它们可以让应用程序以最高效的方式利用默认任务调度程序的特性。

在本书中，局部队列中的任务都被称作子任务。使用这个术语的原因是因为大部分放在局部队列中的任务都是在执行其他任务的用户委托时创建的。

3.6.6 子任务的内联执行

通常第二个任务必须等第一个任务完成后才能执行。如果第二个任务并未开始执行，你可能觉得执行第一个任务的线程就会被阻塞，直到第二个任务最终开始运行并完成。如果第二个任务处在队列中一个不幸的位置，那么程序不知道会等待多长时间，而且在极端情况下甚至会导致死锁（如果其他工作线程都没有空闲的话）。

幸运的是，TPL 能够检测出第二个任务有没有开始执行。如果第二个任务没有开始运行，那么默认任务调度程序有时会直接在第一个任务的线程上下文内执行它。这种技术被称作内联执行，它让一个可能会被阻塞的线程得到再利用。它也排除了死锁的可能性（由于线程饥饿）。另外，内联执行让紧急任务“走捷径”，可以减少总体延迟。

.NET Framework 4 中的默认任务调度程序会内联挂起的子任务，如果这个子任务所在的局部队列对应的工作线程调用了 `Task.Wait` 或 `Task.WaitAll` 方法。内联也可以应用在 `Parallel` 类的方法创建的任务上，如果这些方法是在一个工作线程内调用的。换句话说，线程池工作线程可以让它创建的任务内联执行。全局队列中的顶层任务不能被内联，使用 `LongRunning` 任务创建选项创建的任务也不能内联其他任务。

默认任务调度程序的内联执行策略是基于 `work-stealing` 队列的同步

需求的。在另一个局部队列中移除或创建一个“被处理过的”任务需要额外的、较大的线程间同步开销。另外，其实一般的应用程序几乎从不需要线程间内联。最常见的代码形式就是将子任务放在执行父任务的线程的局部队列中。

3.6.7 线程注入

.NET 线程池自动管理池中的工作线程的数量。它根据内置的试探法添加或删除线程。.NET 线程池使用两个主要的机制来注入线程：一种是避免饥饿的机制，如果队列中的任务没有启动进程，它就会添加工作线程；另一种是爬山试探法，它会试着使用尽可能少的线程达到最大的吞吐量。

避免饥饿法的目的是防止死锁。当工作线程在等待一个同步事件，但这个事件只能被线程池的全局或局部队列中的一个被挂起的工作项目触发的时候，这样的死锁就会产生。如果工作线程的数量是固定的，而且这些线程都被这样阻塞了，那么系统就再也不能运转下去。添加一个新的工作线程能解决这个问题。

爬山试探法的一个目标是在线程被 I/O 或其他占用了处理器的等待条件阻塞时提高内核的利用率。默认情况下，线程池中对应着每个内核都有一个工作线程。如果其中一个线程被阻塞了，那么就可能有一个内核没有被充分利用，这取决于这个计算机的总体工作量。线程注入逻辑不会区分一个线程是被阻塞了还是在执行一个较长的处理器密集型操作。因此，当线程池的全局或局部队列包含了挂起的工作项目时，要花费较长时间的(超过半秒)活动工作项目会触发新线程池工作线程的创建。

.NET 线程池可以在每完成一个工作项目或每隔 500 毫秒时进行线程注入(以二者中时间居短者为准)。线程池会在此时根据之前的线程数量的变化添加线程(或者移除线程)。如果添加线程可以增加系统吞吐量，那么线程池就会添加更多线程；否则便减小工作线程的数量。这种技术就叫做爬山试探法。

因此，单个任务不要太长，其中一个原因就是为了避免“饥饿检测”，另外一个原因是为了让线程池有更多机会通过调整线程数量来提高吞吐量。单个任务耗时越短，线程池就能越频繁地测试吞吐量并相应地调整线程数量。

我们考虑一个极端的情况。假设有一个复杂的财务模拟作业，它包含 500 个处理器密集型操作，平均每个操作都需要 10 分钟才能完成。如果在全局队列里面为每个操作创建一个顶层任务，会发现大约 5 分钟过后，线程池中的工作线程数量增加到 500。这是因为线程池认为所有任务都阻塞了，所以会以大约每秒两个线程的速度添加新线程。

500 个工作线程会产生什么问题呢？原则上来说，如果有 500 个内核和海量系统内存可以让它们利用，那就没什么。实际上，这就是并行计算的长远目标。然而，如果计算机没有那么多内核，那么大量线程都在竞争时间片。这种情况被称作处理器超额申请。允许大量处理器密集型线程在单个内核上竞争时间片会增大切换上下文的花费，这将严重减小系统整体吞吐量。即便还没有用完内存，这种情况下的性能也会比串行计算要差非常多。（每次上下文切换会花费 6000~8000 个处理器周期。）上下文切换的开销并不是所有花费的唯一源头，.NET 中的每个托管线程占用了一兆字节的栈空间，不管这部分空间有没有被用来执行当前的函数。创建一个新线程大约需要 200 000 个 CPU 周期，而撤销一个线程大约需要 100 000 个周期。这些都是高开销的操作。

只要任务不会花费太长时间，线程池的爬山算法最终会发现它有太多的线程并会进行消减。但是，如果确实有任务会占用工作线程太多时间，这种情况下系统会摆脱线程池的试探法，到时候你将从下面的选项中做出选择。

第一个选项是将应用程序分解成较小的任务，小到能够很快执行完毕，以便让线程池成功地控制线程的数量，从而优化系统吞吐量。

第二个选项就是实现自己的任务调度程序对象，让它不进行线程注入。如果任务的执行时间很长，你并不需要一个高度优化的任务调度程序，因为调度的开销和执行这个任务的时间比起来是可忽略的。MSDN 开发者程序中有一个简单的任务调度程序的实现，它限制了最大并发度。欲查看更多信息，请阅读 3.8 节推荐的参考资料。

最后一个选项就是使用 `SetMaxThreads` 方法来配置 `ThreadPool` 类的工作线程数量的上限，一般设置成内核的个数（也就是 `Environment.ProcessorCount` 属性）。这个上限将会对整个进程起作用，包括所有的 `AppDomain`。

如果线程池的工作线程都在等待工作项目运行，`SetMaxThreads` 方法可能会导致死锁。请务必小心使用。

3.6.8 绕过线程池

如果不想让一个任务使用线程池中的工作线程，可以在该线程池之外新建一个专用线程。要做到这点，需要将 `LongRunning` 任务创建选项添加为 `Task` 类的 `Factory` 成员的一个方法的参数。这个选项主要被用在含有长时间 I/O 处理的任务和充当后台辅助程序的任务上。

绕过线程池的一个缺点就是，不像被线程池的线程注入逻辑创建的工作线程那样，使用 `LongRunning` 选项创建的线程不能让它的子任务内联执行。

3.7 练习

1. 本章中的图像混合的例子使用了任务并行性：不同的任务处理不同的图层。图像处理中有一种典型的策略使用了数据并行性：用相同的计算对一张图片的不同部分或不同的图片进行处理。是否有一种方法可用于在图像混合的例子中使用数据并行性呢？如果有，那么它跟这里讨论的任务并行性比起来有什么优点和缺点？
2. 在图像混合的例子中，图像处理方法 `SetToGray` 和 `Rotate` 都是无返

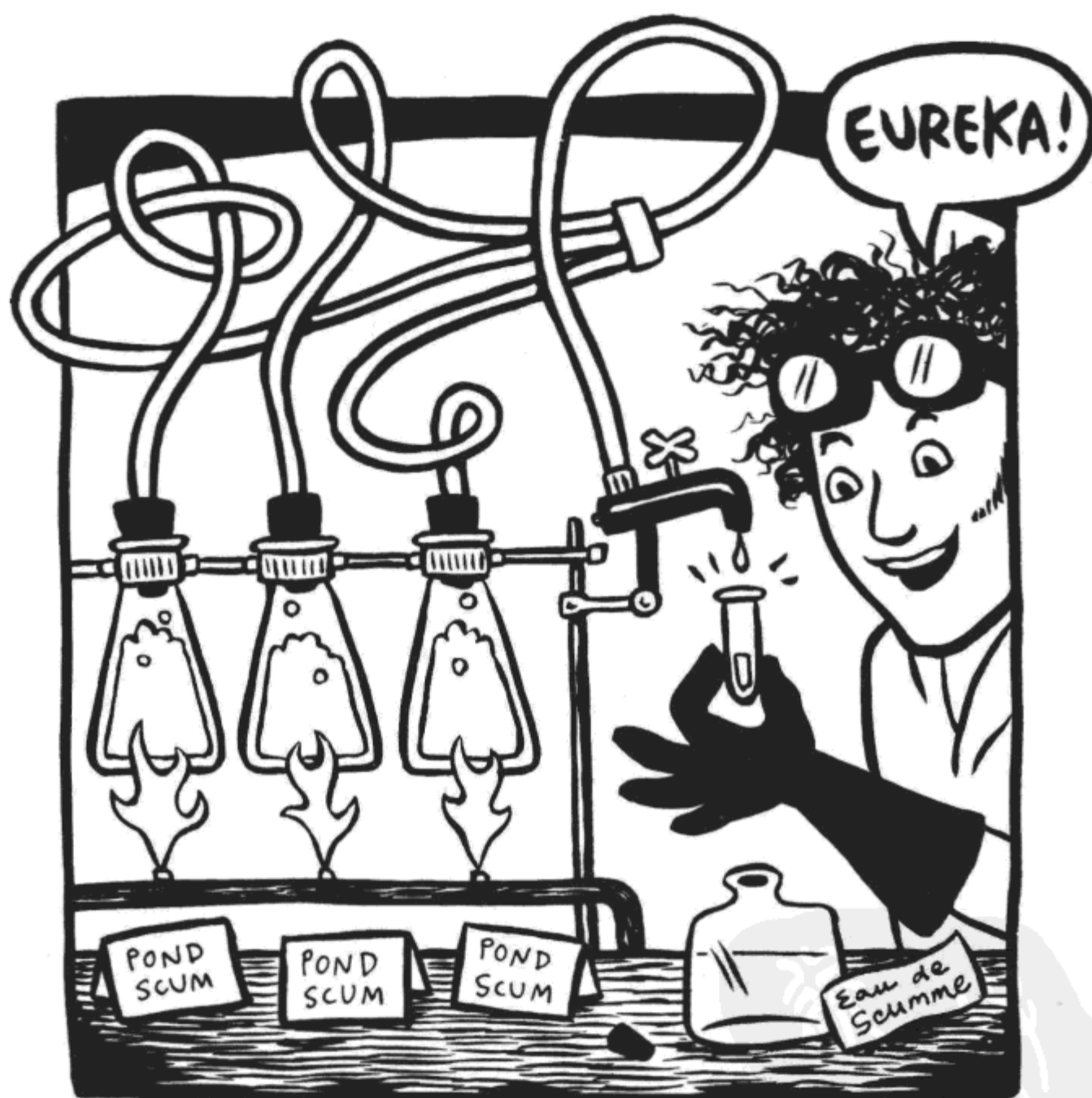
返回值的方法，但它们却通过更新它们的第二个参数来保存结果。它们为什么不直接返回结果？

3. 图像混合的例子中使用了 `Task.Factory.StartNew`，如果其中一个并行任务抛出异常会怎样？使用 `Parallel.Invoke` 方法又会怎样？

3.8 扩展阅读

Leijen 等人的著作讨论了设计根据，包括调度和 work stealing。Hoag 提供了一个详细的有关任务创建选项的论述。MSDN 上的 `ParExtSamples` 是微软的并行示例代码包。MSDN 上的指引性文章可以帮你编写自定义的任务调度程序。MSDN 上的 `Task Scheduler Event` 页面提供了更多有关未观测到的任务异常的信息。

- Leijen D, Schulte W, Burckhardt S. The Design of a Task Parallel Library. In: Arora S, Leavens G T, ed. OOPSLA 2009: Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. ACM, 2009. 227-242.
- Hoag J E. A Tour of Various TPL Options. 2009-04. <http://blogs.msdn.com/b/pfxteam/archive/2010/04/19/9997552.aspx>.
- `ParExtSamples` 软件. Samples for Parallel Programming with the .NET Framework 4. <http://code.msdn.microsoft.com/ParExtSamples>.
- How to: Create a Task Scheduler that Limits The Degree of Concurrency. <http://msdn.microsoft.com/en-us/library/ee789351.aspx>.
- .NET Framework Class Library. Task Scheduler Events. http://msdn.microsoft.com/en-us/library/system.threading.tasks.taskscheduler_events.aspx.



第 4 章 并行合并计算

第 2 章阐述了如何使用并行技术将同一个独立操作应用于不同输入值中。然而，不是所有的并行循环都有可以独立运行的循环体。例如，一个计算总和的循环没有独立的步骤。所有的步骤都用单个变量来累计它们的结果，这个变量表示当前步骤计算的总和。这个累计值是一个合并计算值。

然而，存在一种方法可以使用循环体实现合并计算操作。这就是并行合并计算模式。

尽管计算总和也是合并计算的一个例子，但是并行合并计算模式比它更常用。该模式对于任意相关的二元运算都是有用的。并行合并计算模式的.NET 实现也希望操作相互间是可换的。

并行合并计算模式使用非共享的局部变量来给出最终结果，这些变量会在计算的最后进行合并。对部分的局部计算值使用非共享的局部变量就是让循环步骤变得相互独立的方法。并行合并计算说明了一个原则，就是通常比起对已有的算法添加同步原子，对自己的算法做些改变会更好。想要了解更多关于这个模式算法方面的信息，请参见 4.4 节。

并行合并计算模式也被称为并行简化模式，因为它将多个输入合并成单个输出。

4.1 基础知识

合并计算的最熟悉的应用就是计算总和。以下是一个计算总和的顺序

并行合并计算模式允许使用多核来计算总和以及其他类型的基于结合运算的累计。

新华书店
PDG

代码。

```
double[] sequence = ...
double sum = 0.0d;
for (int i = 0; i < sequence.Length; i++)
{
    sum += Normalize(sequence[i]);
}
return sum;
```

这是一个典型的顺序 for 循环。在这个例子和接下来的例子中，**Normalize** 是一个由使用者提供的方法，它以某种方式将输入值进行转换，例如将输入值转变成相应的度数。其结果就是转换的值的总和。

微软 .NET Framework 语言集成查询(LINQ)提供了一个简单的方法来展示这种类型的合并计算。C#、F#之类的编程语言和 Visual Basic 开发系统为 LINQ 提供了专门的语法。接下来的 LINQ 表达式计算了总和。

```
double[] sequence = ...
return (from x in sequence select Normalize(x)).Sum();
```

LINQ 表达式是一个顺序操作，它的性能可以同之前的顺序 for 循环例子相当。

将一个 LINQ-to-Objects 表达式转换为一个并行查询是非常简单的。以下代码给出了这样一个例子。

```
double[] sequence = ...
return (from x in sequence.AsParallel()
        select Normalize(x)).Sum();
```

如果调用 **AsParallel** 扩展函数，那么就是在命令编译器去编译 PLINQ 而不是 LINQ。程序会对这个表达式使用所有扩展查询操作的并行版本。**Sum** 扩展函数执行了查询操作并(暗中并行地)计算出所选择的值转换后

的总和。PLINQ 的介绍可以参见第 2 章。

这个例子使用加法作为基本的合并运算操作，但是还有其他操作可用。例如，PLINQ 已经构造了标准查询操作来实现元素个数的计算及平均值、最大值和最小值的计算。PLINQ 也有操作来实现新建和合并集合(重复删除、合并、相交和差)、序列转换(级联、筛选和分割)和聚合(投影)。这些标准查询操作对完成很多类型的合并运算任务已经足够了，而且有了 PLINQ，它们可以充分地利用多核计算机的硬件资源。

如果 PLINQ 的标准查询操作中仍没有你想要的，那么也可以使用 **Aggregate** 的扩展函数来定义自己的合并算法操作。以下就是一个例子。

```
double[] sequence = ...  
return (from x in sequence.AsParallel()select Normalize(x))  
        .Aggregate (1.0d, (y1, y2) => y1 * y2);
```

这段代码显示了 **Aggregate** 扩展函数的一个重载版本。它将使用者提供的转换方法应用到每个输入序列元素，然后返回一个转换值的数值结果。

当需要将并行合并计算模式应用到 .NET 应用程序中时，PLINQ 通常是最为推荐的方法。其声明性质使得它相比其他方法更少出现错误，而且它在多核电脑上的性能胜过其他方法。用 PLINQ 执行并行合并计算不需要在代码上加锁。相反，使用 PLINQ，所有的同步运算都会发生在内部。

如果 PLINQ 不能满足需求或者更喜欢非声明式的编码，也可以使用 **Parallel.For** 或者 **Parallel.ForEach** 来执行并行合并计算模式。**Parallel.For** 或者 **Parallel.ForEach** 方法比起 PLINQ 需要更复杂的代码。例如，**Parallel.ForEach** 方法需要代码包含同步原子来执行并行合并计算。想要了解更多的例子和信息，请参见 4.3.1 节。

4.2 示例

合并运算不是只应用于数字值。这是个相当常用的模式，经常出现在很多应用关系中。接下来的例子展示了被称为映射/简化的并行合并计算形式是如何应用于合并非标量数据类型的。

这个例子与社交网络服务有关，在这个网站里，会员可以指定其他会员成为好友。这个网站通过识别其他会员是不是该会员好友的好友来推荐新的好友给每个会员。为了限制推荐的数目，服务器只推荐彼此间拥有最大数目的共同好友的会员。这些会员可以由独立的并行操作识别，然后由合并运算操作将他们排名，并选择要推荐的会员。

下面介绍的是数据结构和算法是如何被应用于推荐服务中的。会员由整数形式的 ID 来识别。一个会员的好友是用他们 ID 的聚集来表示。这个聚集是一个集合，因为每个元素(好友的 ID)只能出现一次，而且元素之间的顺序是不重要的。例如，某个会员的 ID 是 0，且他有两个好友，ID 分别是 1 和 2，那么他们的关系可以写成：

```
0->{1,2}
```

社交网络库为每个会员存储了一个记录。为了给每个会员推荐好友，推荐服务必须考虑这个会员的好友记录，同时也要考虑该会员好友的好友记录。例如，为了给 0 号会员推荐好友，需要考虑的相关记录在库中表示如下：

```
0->{1,2}
1->{0,2,3}
2->{0,1,3,4}
```

由上可以看到，服务器应该推荐 3 号和 4 号会员给 0 号会员，因为他们出现在 1 号会员和 2 号会员的好友记录里，且 1 号会员和 2 号会员已经

是0号会员的好友。另外,相比4号会员,推荐服务应该优先推荐3号会员,因为3号会员是0号会员所有好友的好友,而4号会员只是0号会员其中一个好友的好友。可以将结果表示如下:

```
{ 3(2), 4(1) }
```

这就意味着3号会员与0号会员有两个共同好友,而4号会员与0号会员只有一个共同好友。这是多重集的一个例子。在一个多重集中,每个元素(这个例子中的3和4)的重数由元素出现在集合中的次数(分别为2和1)决定。多重集中元素间的顺序是不重要的。

推荐服务使用了映射/简化(map/reduce)方法,这个方法有几个步骤。第一步,也就是映射阶段,推荐服务新建一个会员集合,该集合可以容纳重复元素,也就是说相同的ID在这个集合中可以出现多次(每个共同好友只出现一次)。第二步就是简化阶段,该服务对这个集合进行合并运算,构造一个多重集合,使得每个候选ID只出现一次,但是ID在第一个集合(共同好友的个数)中是与重数相关的。同样还有后续操作步骤,就是服务器根据候选ID的重数将它们排序,并只选出拥有最大重数的ID。

映射/简化的一个重要特点就是映射阶段的结果集合中的项与简化阶段的结果相一致。因为简化阶段使用了多重集,因此,映射阶段不是只产生ID的集合,而是产生一个多重集的集合,每个多重集包含一个候选ID及其重数。在这个例子中,映射阶段的输出是一个由两个多重集组成的集合。里面的会员是3号和4号会员。

```
{ 3(1) }, { 3(1), 4(1) }
```

这里,第一个多重集包含了1号会员的好友,第二个多重集包含了2号会员的好友。

映射/简化的另一个重要特点是,在简化阶段中的合并运算是通过将二元运算应用到由映射阶段产生的元素组合来执行的。在这个例子中,需要的操作是多重集的合并,就是通过将元素汇总并增加它们的重数来合并

两个多重集。应用多重集合并运算来合并之前集合中的两个多重集的结果如下：

```
{ 3(2), 4(1) }
```

既然只有一个多重集了，那么简化阶段就完成了。通过不断地应用多重集合并运算，简化阶段可以将任意多个多重集合并成一个多重集。

以下是顺序实现代码。

```
public IDMultisetItemList PotentialFriendsSequential(
    SubscriberID id,
    int maxCandidates)
{
    //映射
    var foafsList = new List<IDMultiset>();
    foreach (SubscriberID friend in subscribers[id].Friends)
    {
        var foafs = subscribers[friend].FriendsCopy();
        foafs.RemoveWhere(foaf => foaf == id ||
            subscribers[id].Friends.Contains(foaf));
        foafsList.Add(Multiset.Create(foafs));
    }

    // 简化
    IDMultiset candidates = new IDMultiset();
    foreach (IDMultiset foafs in foafsList)
    {
        candidates = Multiset.Union(foafs, candidates);
    }

    // 后续操作
    return Multiset.MostNumerous(candidates, maxCandidates);
}
```

在映射阶段，这个代码在会员好友中连续地循环，并创建候选会员(foaf 是好友的好友)的多重集的集合。在简化阶段，这个代码在这些多重集中连续地循环，并使用合并运算将它们合并。如果该代码在上述例子中的几个会员中循环，id 变量的值是 0，而 subscribers[id].Friends 是 {1,2}。当映射阶段完成时，foafsList 是 { 3(1) }, { 3(1), 4(1)}；当简化阶段完成时，candidates 是 { 3(2),4(1) }。

多重集的合并是可互换的：其结果不依赖于变量的顺序。多重集的合并也是相关的：如果对多个集合不断地进行合并运算，最后将它们合并成一个集合，那么最终结果不会依赖于合并运算的顺序。如果合并函数是不相关的，那么就不可能以并行的方式得到相同的结果。如果合并函数不是可互换的，那么就大大降低了并行的可能性。

严格来说，浮点数在数学上既不是可互换的，也不是可合并的。浮点的或者双精度的并行计算的运行结果可能每次会稍有不同。

以下就是如何使用 PLINQ 将映射/简化应用于社交网络例子的代码。

```
public IDMultisetItemList PotentialFriendsPLinq(SubscriberID id,
                                                int maxCandidates)
{
    var candidates =
        subscribers[id].Friends.AsParallel()
            .SelectMany(friend => subscribers[friend].Friends)
            .Where(foaf => foaf != id &&
                !(subscribers[id].Friends.Contains(foaf)))
            .GroupBy(foaf => foaf)
            .Select(foafGroup => new IDMultisetItem(foafGroup.Key,
                                                    foafGroup.Count()));
    return Multiset.MostNumerous(candidates, maxCandidates);
}
```

回顾映射/简化过程，独立的并行运算(映射阶段)是紧跟在合并运算(简化阶段)之后的。在映射阶段中，并行运算在 0 号会员的所有好友中重复执行。映射阶段是通过 SelectMany 方法实现的，它查找 0 号会员的每

个好友的所有好友，而 `Where` 方法通过移除 0 号会员和他的好友来避免重复推荐。映射阶段的输出是候选 ID 的一个集合，包含重复 ID。简化阶段是由 `GroupBy` 方法实现的，它将重复的 ID 聚集成组，`Select` 方法将每个组转换成多重集的一项，将候选 ID 与重数(或 `Count`)关联起来。`return` 语句执行了最后的后续操作，选出拥有最大重数的候选 ID。

当映射/简化阶段用 PLINQ 操作时，不需要逐行编译 `foreach` 循环。在 PLINQ 例子中，映射阶段的输出不是多重集的集合，而是重复 ID 的集合。多重集直到简化阶段才能形成。

这个例子的在线代码资源也包含了用 `Parallel.ForEach` 方法实现的映射/简化。

4.3 变化形式

本节包含了并行合并计算模式的常见变化形式。

4.3.1 使用并行循环进行合并计算

`Parallel.ForEach` 和 `Parallel.For` 方法都有重载版，它们可以实现并行合并计算模式。以下是一个例子。

```
double[] sequence = ...
object lockObject = new object();
double sum = 0.0d;

Parallel.ForEach(
    // 合并值
    sequence,

    // 局部初始化部分结果
    () => 0.0d,
```

```
// 循环体
(x, loopState, partialResult) =>
{
    return Normalize(x) + partialResult;
},

// 每个局部循环的最后步骤
(localPartialSum) =>
{
    // 连续访问单独的共享结果
    lock (lockObject)
    {
        sum += localPartialSum;
    }
});
return sum;
```

Parallel.ForEach 根据需要的并行度来划分输入，并创建并行任务来执行循环。每个并行任务都有不能与其他任务共享的工作状态。循环体中，只有任务的工作状态会更新。换句话说，循环体累计的数值首先会传送到部分和中，而不是直接传送到总和中。当每个任务完成时，所有的部分和都会累加到最终总和中。这些步骤的图解详情，请参见 4.4 节的图 4.1。

以下就是这个例子中用到的 **Parallel.ForEach** 的重载版本。

```
Parallel.ForEach<TSource, TLocal>(
    IEnumerable<TSource> source,
    Func<TLocal> localInit,
    Func<TSource, ParallelLoopState, TLocal, TLocal> body,
    Action<TLocal> localFinally);
```

Parallel.ForEach 中有 4 个变量。第一个变量是将被合并的数值。第二个变量是一个委托，它返回合并的初始值。第三个变量也是一个委托，它将某个输入值与上一次迭代中得到的部分和相加。

基于并行任务库(TPL)的 `Parallel.ForEach` 实现创建了一些任务来执行并行循环。当循环中创建了很多任务,那么将会得到每个任务的一个部分结果。工作任务的个数是由基于试探法的并行循环的实现来决定的。一般来说,每个可用的核至少会有一个工作任务,但是不能有更多的工作任务了。当循环进行时, TPL 有时会撤销一些任务进而创建新的任务。

当并行循环将要完成时,它会将所有工作任务中的部分和合并起来得到最终结果。`Parallel.ForEach` 中的第四个变量是一个执行合并操作的委托。每个工作任务只调用一次这个委托。委托中使用的参数就是任务所累计的部分和。这个委托锁住了共享结果变量,并将结果与部分和相加。

这个例子利用了 C#锁关键字来顺序地访问并发线程所共享的变量。还有其他的同步技术也可以用于这种情况,但是这些技术超出本书的论述范围。要意识到锁是相互协作的;也就是说,除非所有可以使用共享变量的线程正确和一致地使用锁,否则就不能保证变量的串行访问。

在 C#中锁的语法如下: `lock(对象){主体}`。对象是识别锁的唯一标识。所有的协作线程必须使用相同的同步对象,因此对象就必须是引用类型,如 `Object` 类型,而不能是数值类型,如 `int` 或 `double`。在 `Parallel.ForEach` 和 `Parallel.For` 中使用锁时,应该创建一个哑元,并设它为可以实现锁功能的一个捕获到的局部变量的值。(一个捕获到的变量是指 `lambda` 表达式所引用的外围环境中的局部变量。)锁的主体是将由锁保护的代码段。该主体的运行时间必须足够小。根据应用程序的不同,由锁对象保护的共享变量也有所不同,如果代码中要访问这些共享变量,编程者必须注意相互间的操作要一致,不能冲突。就这个例子而言,锁对象保证了可以串行访问 `sum` 变量。

应该在代码中记录由带注解的锁所保护的变量。这个很容易犯错。

当一个线程遇到一个锁语句,那么它会尝试获得该锁及其同步对象。一次只能有一个线程可以拥有这种锁。如果另一个线程已经得到了该锁但是还没有释放它,那么当前线程会阻塞,直到可以得到锁为止。当多个线程竞争同一个锁时,它们得到锁的顺序是不能预测的;顺序未必是

FIFO(先进先出)。在某个线程成功地得到锁之后，它会在主体内执行锁语句。当该线程退出锁的主体(不管是正常退出还是抛出异常)时，它就会释放锁。想要了解更多有关锁的知识，请参见 4.4 节。

`localFinally` 这个委托会在每个任务结束前执行一次。因此，所需要的锁的个数会和用于执行并行循环的任务数相等。你无法预测会有多少个任务被用到。例如，你不能假设任务的个数不会超过你提供的 `MaxDegreeOfParallelism` 选项(或者它的默认值)。其原因是并行循环有时会在执行循环期间关闭某些任务，并继续执行新任务。这种可控制的灵活合作在某些情况下有利于通过允许其他任务运行来减少系统的延时。它也有助于预防线程池中的工作线程数目的增长超过预期范围。你不能认为累加器的状态是严格的“线程级”的。事实上，它是“任务级”的。(线程级和任务级的差别通常不会影响一个程序的逻辑部分，因为一个任务从头到尾只在一个线程上完成。)

为了对比，下面给出一个用于这类并行模式的 PLINQ 版例子。

```
double[] sequence = ...  
return sequence.AsParallel().Select(Normalize).Sum();
```

4.3.2 使用范围分割器进行合并计算

当需要执行多次迭代，但每个循环体的工作量不大时，`Parallel.ForEach` 的开销可能比执行循环体的开销更大。

假若遇到这种情况，有时候在循环中使用 `Partitioner` 对象会更有效。`Partitioner` 对象用于在 `Parallel.ForEach` 中嵌入一个连续的 `for` 循环，并且可以减少 `Parallel.ForEach` 循环的迭代次数。一般来说，为了决定是否使用 `Partitioner` 对象，必须清楚了解程序的结构。

以下是一个例子。

```
double[] sequence = ...
```



```
object lockObject = new object();
double sum = 0.0d;
var rangePartitioner = Partitioner.Create(0, sequence.Length);
Parallel.ForEach(
    // 输入区间
    rangePartitioner,

    // 局部初始化部分和结果
    () => 0.0,

    // 每个区间的循环体
    (range, loopState, initialValue) =>
    {
        double partialSum = initialValue;
        for (int i = range.Item1; i < range.Item2; i++)
        {
            partialSum += Normalize(sequence[i]);
        }
        return partialSum;
    },

    // 每个局部内容的最终步骤
    (localPartialSum) =>
    {
        // 使用锁来连续访问共享结果
        lock (lockObject)
        {
            sum += localPartialSum;
        }
    });
return sum;
```

这个代码与 4.3.1 节的例子很像。它们之间的主要区别在于 **Parallel.ForEach** 循环使用了一系列由 **Partitioner** 对象产生的索引间隔而不是个别值来作为

它的输入。这样可以避免一些调用委托函数所涉及的开销。只有当每一步的工作量很小，且步骤非常多时，它才有比较明显的优势。

以下是所使用的 **Parallel.ForEach** 函数的重载版本的声明。

```
Parallel.ForEach<TSource, TLocal>(  
    Partitioner<TSource> source,  
    Func<TLocal> localInit,  
    Func<TSource, ParallelLoopState, TLocal, TLocal> body,  
    Action<TLocal> localFinally);
```

4.3.3 使用带有范围选择的 PLINQ 合并计算

PLINQ Aggregate 的扩展函数包括适用于普遍的并行合并计算模式的程序的重载版。以下是一个财务仿真程序中的例子。该函数重复仿真测试，并将其结果合并到一个柱状图中。这个代码有两个依赖关系是必须处理的。也就是部分和的累加值在结果柱状图中的显示和 **Random** 类的实例 (**Random** 的实例不能在多线程中共享)。

```
int[] histogram = MakeEmptyHistogram();  
  
return ParallelEnumerable.Range(0, count).Aggregate(  
    // 1- 创建一个空的局部累加器对象  
    // 该对象包括所有任务局部的状态  
    () => new Tuple<int[], Random>(  
        MakeEmptyHistogram(),  
        new Random(SampleUtilities.MakeRandomSeed())),  
    // 2- 运行仿真，将结果加到局部累加器中  
    (localAccumulator, i) =>  
    {  
        // 每次迭代得到下一个随机值  
        var sample = localAccumulator.Item2.NextDouble();  
  
        if (sample > 0.0 && sample < 1.0)
```

```

{
    // 执行一个样本值的仿真测试
    var simulationResult =
        DoSimulation(sample, mean, stdDev);

    // 将结果加到局部累加器的柱状图中
    int histogramBucket =
        (int)Math.Floor(simulationResult / BucketSize);
    if (0 <= histogramBucket && histogramBucket < TableSize)
        localAccumulator.Item1[histogramBucket] += 1;
}
return localAccumulator;
},

// 3- 结合局部结果使其配对
(localAccumulator1, localAccumulator2) =>
{
    return new Tuple<int[], Random>(
        CombineHistograms(localAccumulator1.Item1,
            localAccumulator2.Item1),
        null);
},

// 4- 从最终配对抽取结果
finalAccumulator => finalAccumulator.Item1
); // 合并计算

```

想要了解并行程
序中 **Random** 类
的统计方面的局
限性，请参见第
3 章。

这个例子的数据来源于 **ParallelEnumerable** 类中 **Range** 静态函数创建的并行查询。以下是在该例中使用的 **Aggregate** 扩展函数的重载版的声明。

```

Aggregate<TSource, TAccumulate, TResult>(
    this ParallelQuery<TSource> source,
    Func<TAccumulate> seedFactory,
    Func<TAccumulate, TSource, TAccumulate> updateAccumulatorFunc,

```

```
Func<TAccumulate, TAccumulate, TAccumulate>  
    combineAccumulatorsFunc,  
Func<TAccumulate, TResult> resultSelector);
```

其中有4个参数。每个参数都是一个委托方法。

第一个参数是建立每个由查询所创建的工作任务的局部状态的委托。只在任务的最开始时调用一次该委托。在这个例子中，该委托返回一个包含两个域的元组(未命名的记录)实例。第一个域是一个空柱状图。它会累计这个任务的局部仿真结果。第二个域是一个 **Random** 类的实例。它是局部状态的一部分，通过在线程中共享实例来确保仿真不会违背 **Random** 类的需求。注意，事实上可以在所创建的对象上存储局部状态的任何形式。

第二个参数是循环体。这个委托在分割区中的每个数据元素中只调用一次。在这个例子中，循环体创建了一个随机样本，并做了仿真实验。接着它将仿真结果分类装进用于柱状图的桶中，将增量分类装进合适的用于任务局部柱状图的桶中。

第三个参数被局部部分和配对组合所调用。该委托将输入的柱状图合并(即将它们对应的桶值相加，并返回一个带总和的新柱状图)。它返回一个新元组。零变元反映了一个现实，那就是随机数值产生器不再是必要的。要统一所有的局部部分和需要尽可能地多次调用合并委托。

第四个参数从最后的合并局部状态对象中选取结果。

这种类型函数适合用于很多使用并行合并计算模式的情况。注意，它的执行不需要锁。

4.4 设计说明

如果比较串行的和并行的合并计算模式，就会发现并行的设计在算法上增加了一个额外的步骤，那就是合并部分和。**Parallel.ForEach** 和

Parallel.For 函数的图解如图 4.1 所示。

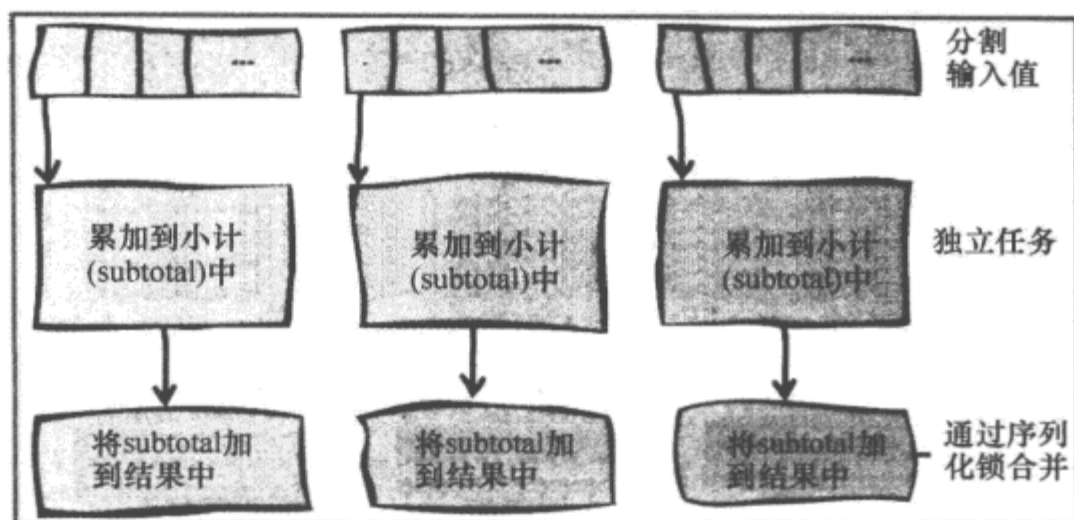


图 4.1 合并计算中用到的 Parallel.ForEach 和 Parallel.For

图 4.1 表明并行循环使用了不共享的局部存储器来存储部分和(在图 4.1 中命名为 subtotal)来代替在一个单独的共享结果上放置累加器。每个任务处理一个输入值的独立分割区。分割区的个数依赖于并行度,这需要使用计算机的可用核。任务在完成分配部分中的累计数值步骤之后,合并它的局部结果为最终的全局结果。所有任务都共享最终结果。锁被用来保证更新同步。

这个方法比较快的原因是因为它需要的锁很少。一般情况下,要处理的元素的个数比任务和分割区的个数要大很多倍。锁的序列化所需要的花费可由很多个别累计操作分别承担。

PLINQ 所用的算法与图 4.1 所示的有点不一样。PLINQ 中的合并计算不需要开发者使用锁。相反,最终的合并步骤可以用一个合并任意两个部分和(也就是任意两个 subtotal)为另一个部分和的二元运算来表示。在接下来的部分和组合中重复以上步骤,最终合并成一个最终结果。PLINQ 方法的一个优势是它不怎么需要同步,因此它就更加具有可扩展性。二元合并不需要系统锁定最终结果;二元运算是可以并行运行的。(这是可能的,因为二元合并计算是关联的,因此这些计算的操作顺序不会影响结果。)

上述讨论表明并行合并计算模式是一个很好的例子来说明为什么当从串行方法转变到并行方法时需要改变算法。

为了使得这个观点更清晰，以下是一个例子，其内容是关于如果只是将锁加到现存的串行算法中，并行合并计算会变得怎么样。其实只需要将顺序 for 循环转换成 `Parallel.For` 循环，并增加一个锁语句。

不能只添加锁来期待得到优良性能，也需要考虑一下算法。

```
// 不要借用这段代码，这个版本的运算运行得比串行版本慢
// 这就说明了为什么不要这么写
double[] sequence = ...
object lockObject = new object();
double sum = 0.0d;
// BUG - 不要这么写
Parallel.For(0, sequence.Length, i =>
{
    // BUG - 不要这么写
    lock (lockObject)
    {
        sum += sequence[i];
    }
});
return sum;
```

如果忘记增加锁语句，那么这段代码就不能成功地在多核计算机上累计错误的总和。如果增加锁语句，这段代码就是正确的。如果运行这段代码，它就会产生预期的总和。然而这种方法并不是最佳选择。这段代码比串行的优化版还要慢很多倍！其性能低下的原因是同步的花费。

相比之下，本章到处可见的并行合并计算模式的例子都将比同样的串行版在多核电脑上运行得更快，而且它们的性能也会根据核数目的比例相应提高。

增加额外的一个步骤可以使得算法运行更快这件事可能初看会与直

觉相悖，但这确实是事实。如果增加额外的工作，而且该工作对预防并行任务中的数据依赖是有效的，那么在性能方面就可以得到改善。

4.5 相关的模式

有很多模式与在一个集合中汇总数据相关。并行计算(也被称为简化)是其中一种。其他的模式还包括浏览模式和打包模式。浏览模式适用于循环中的每个迭代依赖于前一个迭代所计算的数据时。打包模式使用并行循环来选择要保留或丢弃的元素。打包操作的结果是原输入的一个子集。这些模式可以结合起来，如同打包和浏览模式一样。更多相关模式的信息参见 4.7 节。

4.6 练习

1. 考虑小型社交网络例子(拥有 0 号、1 号和 2 号会员)。数据之间存在的约束是什么？在简单代码中这些约束条件是如何被观测到的？
2. 在社交网络例子中，候选会员的多重集由无序变为一个根据共同好友个数排序的有序多重集，接着选出最前面 N 个候选会员，在这两个步骤后还有几个后续操作步骤需要完成。某些或者全部的后续操作可以合并到简化阶段中吗？请用 PLINQ 和 `Parallel.ForEach` 来回答。
3. 在映射/简化的参考标准(参见 4.7 节)中，映射阶段执行一个映射函数，该函数接收一个输入对，并输出一串键/值对作为中间结果。同一个主键的所有键值对都会被传递到简化阶段中。简化阶段执行一个简化函数，将所有重复的中间键的值合并成一个尽可能小的值的集合。这些函数可以被表示为：`map (k1,v1) -> list(k2,v2)` 和 `reduce(k2,list(v2))->list(v2)`。在社交网络例子中， $k1$ 、 $v1$ 、 $k2$ 和 $v2$

是什么类型？映射函数和简化函数是什么？请用 PLINQ 和 Parallel.ForEach 来回答。

4.7 扩展阅读

MSDN 提供了 LINQ 和 PLINQ 的标准查询操作说明。Duffy 的书展示了同步技术的全套方案。McCool 讨论了相关模式的模板模式、浏览模式和打包模式。关于映射/简化的参考标准，Dean 和 Ghemawat 的论文中有说明。Toub 阐述了其他算法的例子，这些算法的步骤之间存在着某些依赖关系。其中包括打包并浏览模式和动态编程。

- Dean J, Ghemawat S. MapReduce: Simplified Data Processing on Large Clusters. In OSDI'04: Sixth Symposium on Operating System Design and Implementation. 2004. 137-150.
- Duffy J. *Concurrent Programming on Windows*. Addison-Wesley, 2008.
- McCool M. Structured Patterns: An Overview. 2009-12. <http://www.ddj.com/go-parallel/article/showArticle.jhtml?articleID=223101515>.
- Toub S. Patterns of Parallel Programming: Understanding and Applying Parallel Patterns with the .NET Framework 4. 2009. <http://www.microsoft.com/downloads/details.aspx?FamilyID=86b3d32b-ad26-4bb8-a3ae-c1637026c3ee&displaylang=en>.
- Standard Query Operators Overview. <http://msdn.microsoft.com/en-us/library/bb397896.aspx>.



第 5 章 future 模式

在第 3 章中，我们学习了如何使用并行任务模式在程序中对控制流分流。本章将探讨怎样通过 future 模式将控制流和数据流整合到一起。

future 是这样的可计算结果的代名词：初始时它们是未知的，但在随后的某个时刻将变得可用。这一结果的计算过程可以与其他计算过程并行执行。future 模式将任务并行机制与我们熟知的参数和返回值的概念联系了起来。如果说第 3 章中描述的并行任务是异步操作，那么 future 便是异步函数。（回想一下，操作没有返回值，但是函数有。）

future 描述了潜在并行性这一概念，该概念在第 1 章中提到过。在硬件设施满足并行所需条件的前提下，通过 future 分解一个顺序操作可以加快处理速度。然而，如果所有的内核都被占用了，future 对并行机制而言就没有多大意义了。

future 是异步函数。

在微软的 .NET Framework 中，future 通过 `Task<TResult>` 类来实现，其中，参数 `TResult` 表示结果的类型。换句话说，在 .NET 中，一个 future 就是一个有返回值的任务。没有必要使用诸如 `Wait` 的方法来明确地等待任务的完成，只需在需要时向任务索要其结果。如果任务已经执行完，就能立即得到结果。如果任务正在执行但还没有完成，需要结果的线程将被阻塞直到任务执行完并得到结果。（当此线程被阻塞时，内核对其他任务来说是可用的。）如果该任务还没有开始，那么可能会在当前线程环境下内联执行。

在 .NET 中，一个 future 就是一个有返回值的任务。future 通过 `Task<TResult>` 类来实现。

.NET Framework 还实现了 future 模式的一个变形，即延续任务。一个 .NET 延续任务是一个当其先行任务完成时自动开始执行的任务。许多情况下，这些先行任务包括一些 future，它们的结果被用作延续任务的输

当特定的先行任务完成时，延续任务会自动执行。

入。一个先行任务可能有不止一个延续任务。

延续任务代表了异步函数的嵌套应用。在某些方面，延续任务有点像回调方法——在这两种情况下，都会注册一个操作，这一操作在将来的某个特定时间将被自动调用。

本章中讨论的 `future` 模式与任务图关系密切。当 `future` 提供了对于其他 `future` 来说是输入结果的值时，这种情形就能用一个有向图来描述。图中的节点代表任务，图的权值代表那些任务的输入值和输出值。

5.1 基础知识

当思考第 3 章中描述的并行任务模式时，将会注意到，许多情况下，大多数任务的目的是计算出一个结果。换句话说，异步操作通常扮演了函数的角色。当然，任务的目的还有许多种，如对于一个数组中的值重新排序，但计算出一个新值这种应用是如此常见，以至于可以支撑起一个适合这种应用的模式。推导纯函数也更加简单，纯函数没有副效应，因此仅仅为结果而存在。随着内核数量的增大，这种简单性变得非常有意义。

5.1.1 `future`

以下的例子来自一个顺序执行方法的方法体。

```
var b = F1(a);  
var c = F2(a);  
var d = F3(c);  
var f = F4(b, d);  
return f;
```

假设 `F1`、`F2`、`F3` 和 `F4` 是处理器密集型函数，它们彼此之间通过参数和返回值进行交互，而不是通过读取和更新共享状态变量来交互。

注意，不要将 `future` 和流水线混淆了。在第 7 章中会介绍，流水线任务也是有向图中的节点，不过连接流水线各阶段的边是并发的队列，它们传送了一连串的数据，就像一条流水作业线或者数据流。而对于 `future` 来说，任务图的节点是由一些单独的值连接起来的，类似于参数和返回值。

同时还假设，你想把这些函数的工作部署到可用的内核中，并且期望无论可用的内核数量是多少，代码都能正确地执行。观察输入和输出，会发现 F1 可以和 F2、F3 并行执行，但是在 F2 完成之前，F3 无法开始。怎么能迅速地发现这些呢？当我们将函数的执行顺序用图来描述时，其中潜在的执行顺序就一目了然了，如图 5.1 所示。

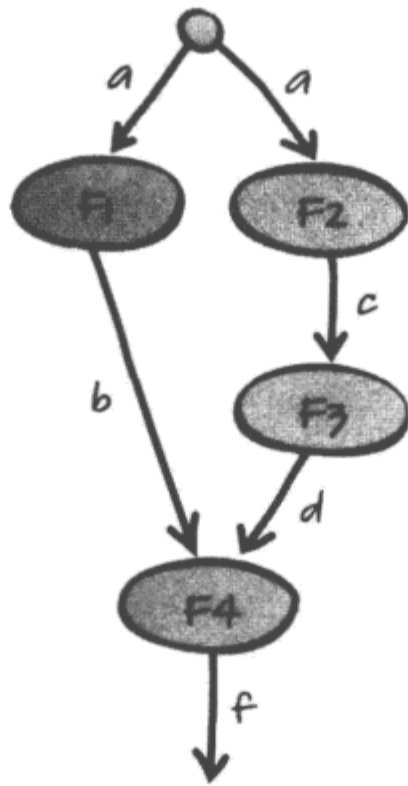


图 5.1 计算 f 的任务图

图的节点代表函数 F1、F2、F3 和 F4。每个节点的传入箭头表示该节点所代表的函数的输入值，传出的箭头表示节点函数计算所得的结果。显然，F1 和 F2 可以同时执行，但是 F3 必须在 F2 之后执行。

以下这个例子演示了怎样为上面的例子创建 `future`。为了简单起见，假设代码中涉及的数值均为整数且已经给出了变量 `a` 的值——可能是当作参数传入的。

```
Task<int> futureB = Task.Factory.StartNew<int>(() => F1(a));  
int c = F2(a);  
int d = F3(c);
```

```
int f = F4(futureB.Result, d);
return f;
```

Result 属性会立即返回一个预先计算好的值，或者等待直到该值可用。

这段代码创建了一个 future，它以异步计算 F1(a) 的值开始。在多核系统中，F1 在当前线程下可以并行执行。这意味着 F2 不必等到 F1 执行完便可开始执行。F4 在获得所需数据后会立即执行。F1 和 F3 的完成顺序是没有影响的，因为 F4 需要 F1 和 F3 的执行结果才能执行。(回想一下，如果 future 值不可用，Result 属性不会返回。)注意对 F2、F3 和 F4 的调用不必封装在一个 future 中，因为仅仅需要一个额外的异步操作就能利用这个例子的并行性。

当然，你可能会像下面这样把 F2 和 F3 放到一个 future 中，这与之前的做法是等价的。

```
Task<int> futureD = Task.Factory.StartNew<int>(
    () => F3(F2(a)));

int b = F1(a);
int f = F4(b, futureD.Result);
return f;
```

这与异步执行任务图中的哪条分支是没有关系的。

future 由 Task<TResult> 类实现，会延迟异常直到读到 Result 属性。

该例中需要注意的一个重点是 future 执行过程中产生的异常是由 Result 属性抛出的。这使得即使是在很多 future 和复杂的延续任务链的情况下，异常也很容易处理。你可以认为 future 返回值或者抛出异常。理论上，这与 .NET 中的函数执行方式非常类似。以下是一个例子。

```
Task<int> futureD = Task.Factory.StartNew<int>(
    () => F3(F2(a)));

try
{
    int b = F1(a);
    int f = F4(b, futureD.Result);
    return f;
}
```

```
}  
catch (MyException)  
{  
    Console.WriteLine("Saw MyException exception");  
    return -1;  
}
```

如果 F2 或者 F3 中抛出类型为 `MyException` 的异常，当读到 `futureD` 的 `Result` 属性时，它将被延迟且重新抛出。在 `try` 代码块中得到 `Result` 属性的值，这意味着异常将在相应的 `catch` 代码块中得到处理。

5.1.2 延续任务

有一种现象很常见，即一个异步操作调用另一个异步操作并将数据传给它。延续任务使得 `future` 之间的依赖关系在运行环境中变得非常明显并且使得调度它们成为可能。这对于内核任务的高效分配是非常有帮助的。

例如，如果用上一节中的 F4 提供的结果来更新用户界面(UI)，可以使用下面这段代码。

```
TextBox myTextBox = ...;  
  
var futureB = Task.Factory.StartNew<int>(() => F1(a));  
var futureD = Task.Factory.StartNew<int>(() => F3(F2(a)));  
  
var futureF = Task.Factory.ContinueWhenAll<int, int>(  
    new[] { futureB, futureD },  
    (tasks) => F4(futureB.Result, futureD.Result));  
  
futureF.ContinueWith((t) =>  
    myTextBox.Dispatcher.Invoke(  
        (Action) (() => { myTextBox.Text = t.Result.ToString(); })  
    ));
```


这段代码将计算过程细分成 4 步。系统理解延续任务与其先行任务之间的依赖关系。它确保延续任务只有在其先行任务执行完之后才能开始执行。

第一步, futureB, 计算 b 的值。第二步, futureD, 计算 d 的值。这两步能并行执行。第三步, futureF, 计算 f 的值。在第一步和第二步执行完成之后, 才能执行第三步。第四步, 通过 F4 得到结果并更新用户界面中的文本框。

ContinueWith 方法创建了一个延续任务, 该延续任务有一个先行任务。对象 Task.Factory 的方法 ContinueWhenAll<TAntecedentResult, TResult>可用于创建基于一个以上的先行任务的延续任务。

5.2 示例: Adatum 金融仪表盘

以下是一个关于 future 模式如何在应用程序中使用的例子。这个例子演示了怎样在应用程序中使用图形用户界面(GUI)运行计算密集型操作。

Adatum 是一个金融服务公司, 它为其雇员提供了金融仪表盘应用程序。这种应用程序通常称为 Adatum 仪表盘, 它使得 Adatum 的雇员能进行金融市场分析。仪表盘应用程序运行在雇员的桌面工作站中。Adatum 仪表盘分析历史数据, 而不是分析实时价格数据流。分析过程需要大量的计算, 而且可能会有一些 I/O 延迟, 因为 Adatum 仪表盘应用程序通过网络搜集多个来源的输入数据。

当该应用程序搜集到市场数据之后, 会将数据集整合到一起。然后, 它会对整合后的市场数据作标准化处理并开始进行分析, 分析的结果就是得到一个市场模型。该应用程序也能对来自美国联邦储备系统的历史数据作同样的处理。当得到历史模型和现实模型之后, 该应用程序就能在两者之间做出比较, 提出诸如“卖出”、“买进”、“持有”等市场建议。可

以用一个图来形象地描述这些步骤，如图 5.2 所示。

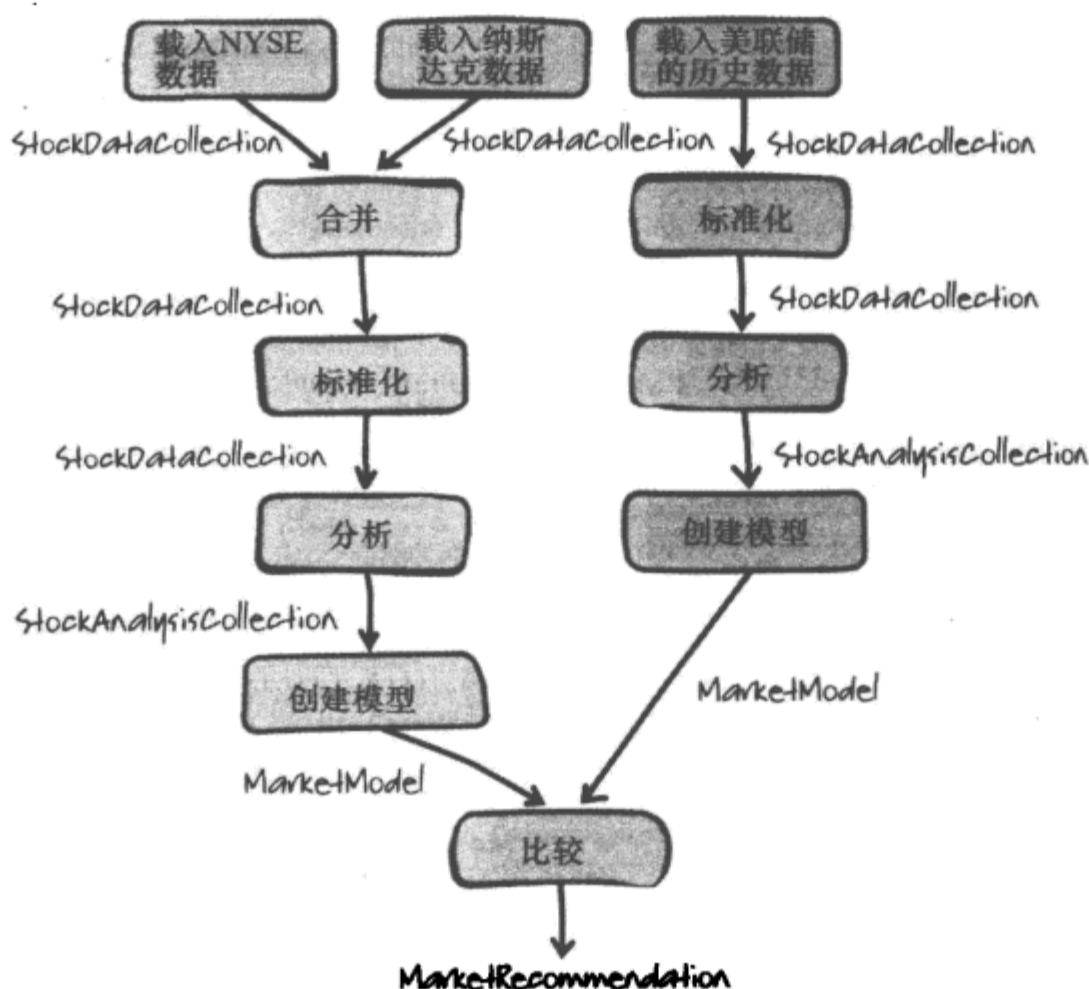


图 5.2 Adatum 仪表盘任务

此图中的任务通过业务对象的特定类型进行交互。Adatum 仪表盘应用程序是通过.NET 类来实现的。

可以下载 Adatum 仪表盘应用程序的源代码，下载地址为 <http://parallelpatterns.codeplex.com>，源代码在 Chapter5\A-Dash 项目中。该应用程序包括 4 部分：业务对象定义、一个分析引擎、一个视图模型和一个用户界面(或者视图)，如图 5.3 所示。

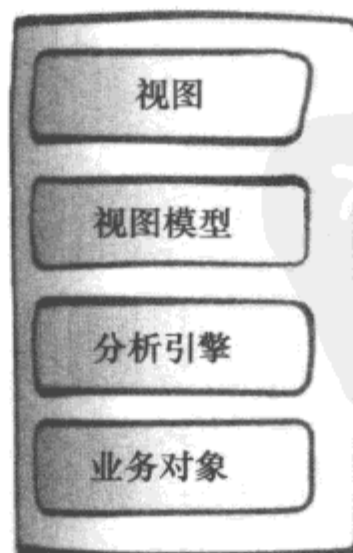


图 5.3 Adatum 仪表盘应用程序

5.2.1 业务对象

Adatum 仪表板应用使用的是不可变的数据类型。当这些类型的对象被创建之后就不能被修改，这使得这些对象非常适用于并行应用程序。

StockDataCollection 类表示一组证券的收盘价时间序列。可以将其假想为一个以股票代码为索引的字典。理论上，数据是每张证券的价格数组。只要股票代码不重叠，都可以合并 StockDataCollection 值。合并操作的结果就是一个新的 StockDataCollection 值，它包含输入的时间序列。

StockAnalysisCollection 类是分析结果，而 MarketModel 类和 MarketRecommendation 类分别是应用程序建模和比较操作产生的结果。MarketRecommendation 类有一个属性用于表示“持有、购买或者卖出”的决定。

更多关于怎样实现自己的不可变类型的知识参见附录 A.4 节。

5.2.2 分析引擎

Adatum 仪表板的 AnalysisEngine 类从所得到的市场数据中生成市场建议。顺序过程代码如下所示：

```
public MarketRecommendation DoAnalysisSequential()
{
    StockDataCollection nyseData =
        LoadNyseData();
    StockDataCollection nasdaqData =
        LoadNasdaqData();
    StockDataCollection mergedMarketData =
        MergeMarketData(new[] {nyseData, nasdaqData});
    StockDataCollection normalizedMarketData =
        NormalizeData(mergedMarketData);
    StockDataCollection fedHistoricalData =
```

```
LoadFedHistoricalData();
StockDataCollection normalizedHistoricalData =
    NormalizeData(fedHistoricalData);
StockAnalysisCollection analyzedStockData =
    AnalyzeData(normalizedMarketData);
MarketModel modeledMarketData =
    RunModel(analyzedStockData);
StockAnalysisCollection analyzedHistoricalData =
    AnalyzeData(normalizedHistoricalData);
MarketModel modeledHistoricalData =
    RunModel(analyzedHistoricalData);
MarketRecommendation recommendation =
    CompareModels(new[] { modeledMarketData,
                           modeledHistoricalData });
return recommendation;
}
```

计算的最终结果是一个 **MarketRecommendation** 对象。每一次方法调用都会返回数据，这些数据变成了执行方法调用操作的输入。用这种方式调用方法时，只能顺序执行。**DoAnalysisSequential** 方法只有在所有它所依赖的操作执行完成时才能返回。

通过 **future** 和延续任务能实现运行步骤的并行版本。代码如下：

```
public AnalysisTasks DoAnalysisParallel()
{
    TaskFactory factory = Task.Factory;
    // ...

    Task<StockDataCollection> loadNyseData =
        Task<StockDataCollection>.Factory.StartNew(
            () => LoadNyseData(),
            TaskCreationOptions.LongRunning);
}
```

```
Task<StockDataCollection> loadNasdaqData =
    Task<StockDataCollection>.Factory.StartNew(
        () => LoadNasdaqData(),
        TaskCreationOptions.LongRunning);

Task<StockDataCollection> mergeMarketData =
    factory.ContinueWhenAll<StockDataCollection,
        StockDataCollection>(
        new[] { loadNyseData, loadNasdaqData },
        (tasks) => MergeMarketData(
            from t in tasks select t.Result));

Task<StockDataCollection> normalizeMarketData =
    mergeMarketData.ContinueWith(
        (t) => NormalizeData(t.Result));

Task<StockDataCollection> loadFedHistoricalData =
    Task<StockDataCollection>.Factory.StartNew(
        () => LoadFedHistoricalData(),
        TaskCreationOptions.LongRunning);

Task<StockDataCollection> normalizeHistoricalData =
    loadFedHistoricalData.ContinueWith(
        (t) => NormalizeData(t.Result));

Task<StockAnalysisCollection> analyzeMarketData =
    normalizeMarketData.ContinueWith(
        (t) => AnalyzeData(t.Result));

Task<MarketModel> modelMarketData =
    analyzeMarketData.ContinueWith(
        (t) => RunModel(t.Result));

Task<StockAnalysisCollection> analyzeHistoricalData =
    normalizeHistoricalData.ContinueWith(
```



```
(t) => AnalyzeData(t.Result));

Task<MarketModel> modelHistoricalData =
    analyzeHistoricalData.ContinueWith(
        (t) => RunModel(t.Result));

Task<MarketRecommendation> compareModels =
    factory.ContinueWhenAll<MarketModel, MarketRecommendation>(
        new[] { modelMarketData, modelHistoricalData },
        (tasks) => CompareModels(from t in tasks select t.Result));

Task errorHandler = CreateErrorHandler(loadNyseData,
    loadNasdaqData, loadFedHistoricalData,
    mergeMarketData, normalizeHistoricalData,
    normalizeMarketData, analyzeHistoricalData,
    analyzeMarketData, modelHistoricalData,
    modelMarketData, compareModels);

return new AnalysisTasks()
{
    LoadNyseData = loadNyseData,
    LoadNasdaqData = loadNasdaqData,
    MergeMarketData = mergeMarketData,
    NormalizeMarketData = normalizeMarketData,
    LoadFedHistoricalData = loadFedHistoricalData,
    NormalizeHistoricalData = normalizeHistoricalData,
    AnalyzeMarketData = analyzeMarketData,
    AnalyzeHistoricalData = analyzeHistoricalData,
    ModelMarketData = modelMarketData,
    ModelHistoricalData = modelHistoricalData,
    CompareModels = compareModels,
    ErrorHandler = errorHandler
};
}
```

并行版本由方法 `DoAnalysisParallel` 实现，它大体上与顺序版本是相似的，最大的不同点是同步函数调用被 `future` 和延续任务取代。该方法返回一个 `AnalysisTasks` 对象，这个对象包含了与每个计算步骤相关的计算任务。`DoAnalysisParallel` 方法立即返回，留下任务继续执行。下一节将描述怎样创建每一个任务。

1. 加载外部数据

需要额外提高并行性时，使用“长期运行”任务，例如有长期运行的 I/O 任务时。

从网络环境中搜集外部数据的方法是持续的、I/O 密集型操作。与其他步骤不同，它们不是处理器密集型的，但是完成它们需要消耗相当长的时间。大部分时间它们都在等待 I/O 操作完成。你会通过一个 `factory` 对象来创建任务，并通过一个参数来表明这些任务持续时间长。这会在系统允许的范围内暂时增加并发性。以下代码显示了外部数据的加载过程：

```
Task<StockDataCollection> loadNyseData =  
    Task<StockDataCollection>.Factory.StartNew(  
        () => LoadNyseData(),  
        TaskCreationOptions.LongRunning);  
  
Task<StockDataCollection> loadNasdaqData =  
    Task<StockDataCollection>.Factory.StartNew(  
        () => LoadNasdaqData(),  
        TaskCreationOptions.LongRunning);
```

注意 `factory` 创建的 `future` 返回类型为 `StockDataCollection` 的值。枚举值 `TaskCreationOptions.LongRunning` 告诉任务库你想要更多的并发。为了防止处理器资源利用不充分，任务库可以选择其他类似的线程运行这些任务。

2. 合并

合并操作的输入数据来自任务 `loadNyseData` 和 `loadNasdaqData`。它是一个基于两个先行任务的延续任务，合并操作代码如下所示：


```
Task<StockDataCollection> mergeMarketData =  
    factory.ContinueWhenAll<StockDataCollection,  
        StockDataCollection>(  
        new[] { loadNyseData, loadNasdaqData },  
        (tasks) => MergeMarketData(  
            from t in tasks select t.Result));
```

当 `loadNyseData` 和 `loadNasdaqData` 完成后，作为参数给出的方法 `MergeMarketData` 被调用了。此时，`tasks` 参数是一个先行任务数组，数组中的内容是任务 `loadNyseData` 和 `loadNasdaqData`。

`MergeMarketData` 方法的输入是一个 `StockDataCollection` 对象数组。LINQ 表达式 `from t in tasks select t.Result` 通过获得每个 `future` 的 `Result` 属性，将 `future` 的输入数组映射到一个 `StockDataCollection` 对象集合中。

3. 标准化

市场数据合并完成后，执行标准化过程。

```
Task<StockDataCollection> normalizeMarketData =  
    mergeMarketData.ContinueWith(  
        (t) => NormalizeData(t.Result));
```

`ContinueWith` 方法创建一个延续任务，该延续任务有一个先行任务。该延续任务通过变量 `mergeMarketData` 从被引用的任务中得到结果值，并调用 `NormalizeData` 方法。

4. 分析和建立模型

当市场数据标准化完成之后，应用程序开始分析过程。该过程的输入为一个类型为 `StockAnalysisCollection` 的对象，并返回一个类型为 `MarketAnalysis` 的对象，操作代码如下所示：

```
Task<StockAnalysisCollection> analyzeMarketData =  
    normalizeMarketData.ContinueWith(  
        (t) => AnalyzeData(t.Result));
```

```
(t) => AnalyzeData(t.Result));

Task<MarketModel> modelMarketData =
    analyzeMarketData.ContinueWith(
        (t) => RunModel(t.Result));
```

5. 处理历史数据

该应用程序还会建立一个关于历史数据的模型。这个模型的建立过程与当前数据模型的建立过程相似。然而，由于这些过程是通过任务执行的，在硬件资源允许的情况下，它们可能并行运行。

6. 模型的比较

以下是比较两个模型的代码。

```
Task<MarketRecommendation> compareModels =
    factory.ContinueWhenAll<MarketModel, MarketRecommendation>(
        new[] { modelMarketData, modelHistoricalData },
        (tasks) => CompareModels(
            from t in tasks select t.Result));
```

这一过程比较当前数据模型和历史数据模型并生成最终结果。

5.2.3 视图和视图模型

Adatum 仪表板是一个基于图形用户界面的应用，使用了模型-视图-视图模型(MVVM)模式。它将并行计算分解成一些子任务，这些子任务的状态都能从图形界面中观察到。更多关于仪表板应用程序与视图模型和视图交互的信息参见附录 A.3 节。

5.3 变化形式

到目前为止，已经看到了使用 `future` 和延续任务来创建任务的最常用的方法。本章将描述使用它们的其他方法。

5.3.1 取消 future 和延续任务

有许多方法来取消 future 和延续任务。可以完全在任务范围内进行取消操作,就像 Adatum 仪表板那样,或者可以在创建任务时传递取消标记。

Adatum 仪表板应用程序支持在用户界面上取消操作。它通过调用 `CancellationTokenSource` 类中的 `Cancel` 方法来完成取消过程。这样的结果是将取消标记的 `IsCancellationRequested` 属性设定为 `true`。

Adatum 仪表板应用程序会在不同的检测点检查这种状况。如果已经提出了取消的请求,则操作被取消。3.3.1 节中描述了一个代码例子。

5.3.2 拥有多个先行任务的情况

如果一个延续任务有多个先行任务,当其第一个先行任务完成时,便可调用该延续任务。这个过程通过对象 `Task.Factory` 中的 `ContinueWhenAny` 方法来实现。当任意一个任务的结果满足条件时, `ContinueWhenAny` 方法将非常有用。例如,可能会有这样一个应用程序,其中的每个任务查询一个提供当地天气预报的 Web 服务。该应用程序会将它得到的第一个结果返回给用户。

5.3.3 使用 .NET 异步调用和 future

任务在某些方面与使用 .NET 异步编程模型 (APM) 模式和 `IAsyncResult` 接口的异步方法是相似的。实际上, .NET Framework 4 中的任务都是 `IAsyncResult` 对象。它们实现这个接口。这一特性可以使你在使用 APM 模式时使用 `Task` 类。

可以将一对使用 `IAsyncResult` 的 `begin/end` 方法转化为任务。对象 `Task.Factory` 的 `FromAsync` 方法具有这种功能。

总之,任务是所有 `IAsyncResult` 的实现中最便于使用的,因为当请求结果时, future 会重新抛出异常。

.NET 中的任务实现 `IAsyncResult` 接口。

5.3.4 消除瓶颈

关键路径的概念与项目管理联系密切。路径是指任意的一个从开始到结束的任务序列。一个任务图可能不止一个路径。例如，观察图 5.2 所示任务图，可以发现该图中有三条路径，这三条路径分别以“载入 NYSE 数据”、“载入纳斯达克数据”和“载入美联储的历史数据”任务开头，均以“比较”任务结束。

一条路径的期限是指该路径上所有任务的执行时间总和。关键路径是指持续时间最长的路径。得到最终结果所需的时间仅与关键路径有关。只要有足够的资源(即可用内核)，非关键路径就不会影响总的执行时间。

如果想使任务图运行得更快，就需要想办法来减少关键路径的持续时间。可以通过更有效率的组织来做到这点。可以将执行最慢的任务分解为若干小任务，这样小任务便可以并行执行。还可以改进一个特别耗时的任务，使它可以在内部使用本书中描述的任何一种模式并行执行。

Adatum 仪表板应用程序并没有给分解执行最慢的任务以使其并行执行这种做法留下多少空间，因为它的路径是线性的。然而，如果分析任务(如图 5.2 所示)消耗时间最多，可以使用并行循环和并行聚集模式来发掘它们之中更多的潜在并行性。任务图保持不变，但是其中的任务却具有了并行性。

5.3.5 运行时修改图

Adatum 仪表板应用程序中的分析引擎代码会生成一个静态的任务图。换句话说，任务图中的依赖关系将直接反映在代码中。通过读懂分析引擎的实现代码，就可以确定存在一定数量的任务，它们之间存在着固定的依赖关系。

在用户界面(UI)层进行分析任务的扩展是一个动态任务创造的范例。通过有计划地添加延续任务，UI 扩展了任务图，而且是在这些任务最初

被创建的上下文环境之外。

动态创建任务也是一种构造算法的方法，这些算法的功能包括排序、查找以及图的遍历。示例参见第 6 章。

5.4 设计说明

在 Adatum 仪表板应用程序中体现的一些决策需要注意。

5.4.1 分解成 future 和延续任务

第一点设计决策是最明显的：Adatum 仪表板应用程序通过 future 和延续任务引进了并行机制。这样做是有道理的，因为问题能被分解成若干操作，而这些操作有着定义完备的输入和输出。

5.4.2 函数式风格

有间接和直接的方法来同步任务间的数据。本章中的例子使用了直接的方法。数据在任务之间以参数传递，这使得数据依赖对于程序员来说非常明显。另外，正如第 3 章所述，任务之间通过共享数据结构来进行交互也可能产生不好的效果。在这种情况下，需要依靠任务的控制依赖来适度阻止。然而，通常，直接数据流没有间接数据流那么容易出错。

可以通过比喻来更好地理解。原则上，并不需要一种支持有返回值方法的编程语言。程序员通常可以使用没有返回值的方法，他们通过全局共享变量来进行数据更新，以此作为应用程序不同组件之间交互的方式。然而，实际上，使用返回值可以在编程过程中少犯错误。程序员使用全局变量比使用返回值更容易犯错误。

同样，对那些通过更新共享全局状态来交互结果的任务来说，future(有返回值的任务)可以在并行编程中减少犯错误的几率。而且，较之需要访问全局状态变量的任务，有返回值的任务需要更少的同步，并且

任务之间通过参数和返回值进行交互的规模会随着内核数量的增大而增大。

有返回值的任务更容易理解。

`future` 促进了一种数据分离的自然方式，该方式与在函数式编程中发现的类似，依赖于输入与输出之间的交互操作。函数程序易于适应多核环境。通常，`future` 与外界交互只能通过返回值。使用不可变的返回值类型也是一种很好的做法。

5.5 相关的模式

有一些模式与 `future` 模式有许多相似之处，但是也有一些很重要的不同点。本节进行了一个简单的比较。

5.5.1 流水线模式

流水线模式将会在第 7 章中进行详细描述。它与任务图在几个方面有些重要的不同。流水线通过队列方式(消息缓存区)专注于数据流，而不是任务依赖。在流水线中，同一个任务针对多个数据项执行。

5.5.2 主/从(master/worker)模式

主/从模式中的任务具有父/子关系，而不是延续任务具备的前身/附属关系。主任务创建工作者任务，为其传递数据，并且等待结果的返回。通常情况下，工作者任务将会对不同的数据执行相同的计算。.NET Framework 4 中的并行循环的实现就使用了主/从模式。

5.5.3 动态任务并行模式

该模式也被称为分治模式，它是第 6 章中的内容。动态任务并行机制在运行过程中以一种类似于递归的方法创建任务树。如果 `future` 是异步函数，动态任务并行机制产生异步递归函数。

5.5.4 离散事件模式

离散事件模式的重点在于任务之间的消息发送。对于任务引发的事件数目以及引发事件的时间是没有限制的。事件也可以在任务之间互相传递；没有前身/依赖关系。通过添加额外的限制条件，离散事件模式能用于实现一个任务图。

5.6 练习

1. 假设以 5.1 节中的第一个例子所用方式来并行化下面这段顺序执行代码。

```
var b = F1(a);  
var d = F2(c);  
var e = F3(b, d);  
var f = F4(e);  
var g = F5(e);  
var h = F6(f, g);
```

画出任务图。为了达到最大程度的并发性，至少需要定义几个 future 呢？能同时运行的 future 的最大数目是多少呢？

2. 修改 CodePlex 中的 Basicfutures 例子，以使其中的一个 future 抛出一个异常。这样将会出现什么现象呢？观察修改后的样本的运行情况。

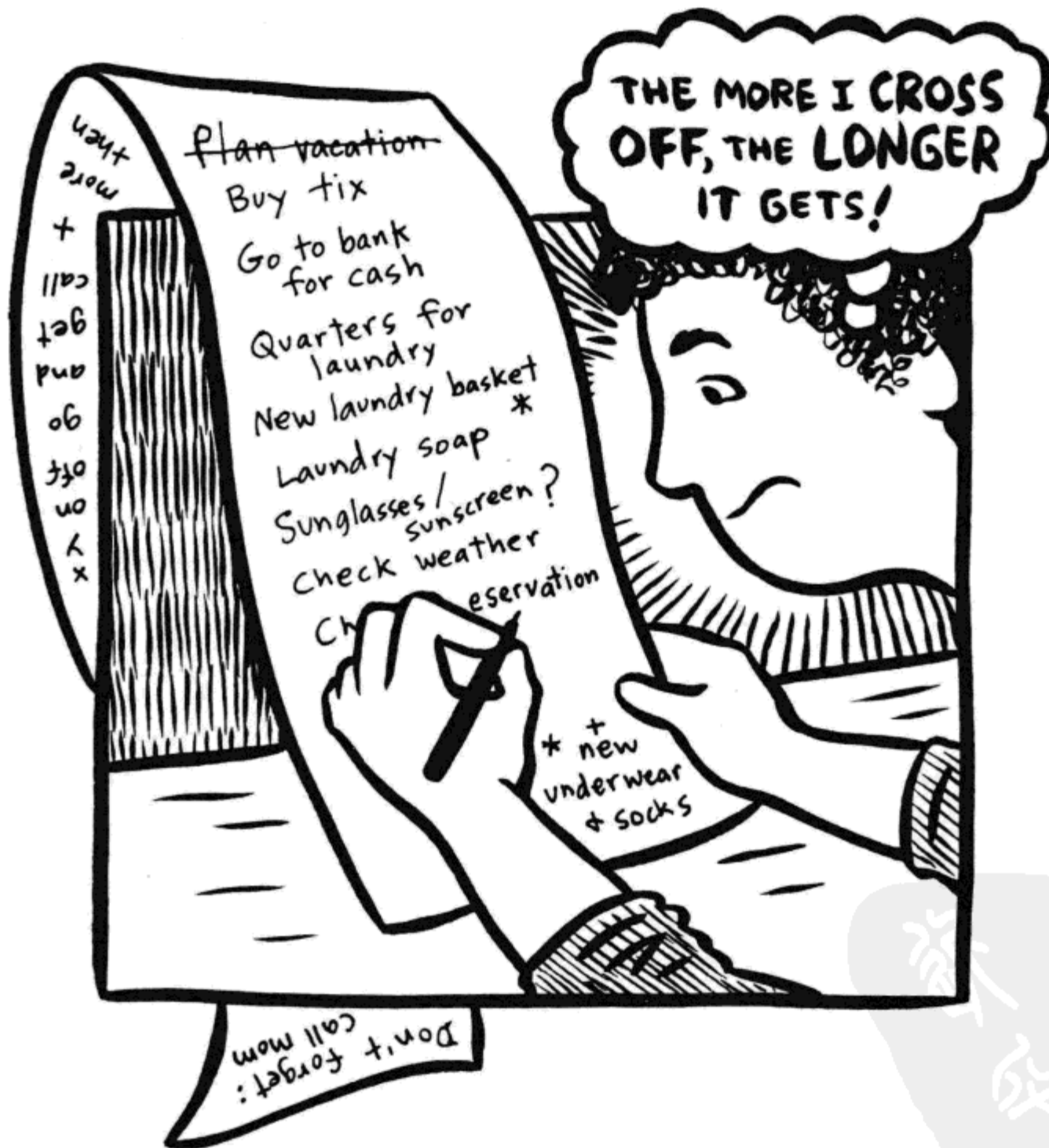
5.7 扩展阅读

Leijen 的论著描述了在任务并行库(TPL)中包含 future 并且引用其他工作的动机，特别是在函数式语言中。NModel 框架提供了一个非可变集

合类型(包括集、包、序列和图)的 C#库。

- Leijen D, Schulte W, Burckhardt S. The Design of a Task Parallel Library. In: Arora S, Leavens G T, ed. OOPSLA, 2009: Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. ACM, 2009. 227 - 242.
- NModel 软件. 2008. <http://nmodel.codeplex.com/>.





第 6 章 动态任务并行

本章涉及的主要内容是在计算过程中动态地向工作队列添加任务，即动态任务并行。动态任务并行的一个简单例子是包含递归算法的顺序程序。

动态任务并行也被称为递归分解或“分治法”。

适合用动态任务并行解决的问题可以被分解为规模更小的相关问题。例如，要计算二叉树的节点数，可以先计算其左右子树的节点数，然后再相加。

动态任务并行和递归类似。为相应的子问题创建子任务。

在数据结构中树和图是动态任务并行应用的典型示例。在地理学或几何学中，一些可以分割的问题也可以应用动态任务并行。

6.1 基础

以下是二叉树的代码。

```
public class Tree<T>
{
    public T      Data { get; set; }
    public Tree<T> Left { get; set; }
    public Tree<T> Right { get; set; }
}
```

如果要对树的数据节点进行操作，需要先访问每个节点，即遍历一棵树。自然首先想到的是递归操作。以下是顺序代码的例子。

```
static void SequentialWalk<T>(Tree<T> tree, Action<T> action)
{
    if (tree == null) return;
    action(tree.Data);
    SequentialWalk(tree.Left, action);
    SequentialWalk(tree.Right, action);
}
```

也可以利用并行任务方式遍历树，代码如下：

```
static void ParallelWalk<T>(Tree<T> tree, Action<T> action)
{
    if (tree == null) return;
    var t1 = Task.Factory.StartNew(
        () => action(tree.Data));
    var t2 = Task.Factory.StartNew(
        () => ParallelWalk(tree.Left, action));
    var t3 = Task.Factory.StartNew(
        () => ParallelWalk(tree.Right, action));

    Task.WaitAll(t1, t2, t3);
}
```

实际上在遍历右子树时可以直接使用当前任务，而不必创建新的任务。但是，不论是遍历右子树还是左子树，都应当确保抛出的异常能被捕获到。了解更多如何捕获未处理任务异常的知识，参见 3.3.2 节。

动态任务并行方式相对于顺序方式的代码而言，可预测性较低。

利用动态任务并行技术执行树的遍历操作时，不必按照预定的顺序访问节点。如果需要采用先序遍历、中序遍历或后序遍历等方式访问节点。请参见第 7 章中谈到的流水线模式。

在本例中，创建的任务数是树中节点数量的三倍，这是相当大的。任务并行库(Task Parallel Library, TPL)就是设计用来处理此类情况的，关于这方面的更多建议，请参考 6.4 节。

6.2 示例

QuickSort 算法是动态并行任务技术的非常典型的例子。QuickSort 算法首先对无序数组进行划分，然后进行排序并重组。以下是其顺序方式的实现。

排序类的应用程序可以充分利用动态任务并行的优势。

```
static void SequentialQuickSort(int[] array, int from, int to)
{
    if (to - from <= Threshold)
    {
        InsertionSort(array, from, to);
    }
    else
    {
        int pivot = from + (to - from) / 2;
        pivot = Partition(array, from, to, pivot);
        SequentialQuickSort(array, from, pivot);
        SequentialQuickSort(array, pivot + 1, to);
    }
}
```

从上面的代码可知，算法并未返回一个有序数组。其 `from` 参数和 `to` 参数定义了待排数组的起止位置。进一步优化代码可知，对于长度很短的数组采用递归算法的效率并不理想。因此对于长度小于 `Threshold` 值的数组，采用非递归的 `InsertionSort` 方法。其中 `Threshold` 是一个全局变量。这种优化对顺序方式和并行方式都适用。

如果数组长度大于 `Threshold` 值，将启用递归算法。每次划分时，选取数组中的中间元素作为枢轴(`pivot`)。Partition 方法将数组中所有较 `pivot` 值小的元素放置在 `pivot` 之前，而将所有较 `pivot` 值大的元素放置在 `pivot` 之后(在此过程中，`pivot` 自身的位置也可能移动)。然后分别在前后两部分

递归调用 `QuickSort` 方法。

以下是并行方式的代码实现。

```
static void ParallelQuickSort(int[] array, int from,
                              int to, int depthRemaining)
{
    if (to - from <= Threshold)
    {
        InsertionSort(array, from, to);
    }
    else
    {
        int pivot = from + (to - from) / 2;
        pivot = Partition(array, from, to, pivot);
        if (depthRemaining > 0)
        {
            Parallel.Invoke(
                () => ParallelQuickSort(array, from, pivot,
                                         depthRemaining - 1),
                () => ParallelQuickSort(array, pivot + 1, to,
                                         depthRemaining - 1));
        }
        else
        {
            ParallelQuickSort(array, from, pivot, 0);
            ParallelQuickSort(array, pivot + 1, to, 0);
        }
    }
}
```

从上面的代码可知，在并行任务中由 `Parallel.Invoke` 执行递归调用。在每次递归中，任务被动态地创建；如果数组很大，则可能创建更多的任务。

并行方式还可以进行一些其他优化。一般而言,创建的任务过多,超出了处理器的执行能力是没有意义的。因此 ParallelQuickSort 算法中定义了 depthRemaining 参数用来限制创建的任务数。depthRemaining 参数随着每次递归调用而递减,只有该参数值大于 0 时才能创建任务。以下代码显示了如何通过处理器数目计算相应的深度(即 depthRemaining 参数)。

```
public static void ParallelQuickSort(int[] array)
{
    ParallelQuickSort(array, 0, array.Length,
        (int) Math.Log(Environment.ProcessorCount, 2) + 4);
}
```

任务的预计执行时间的相近度是确定任务数目的相关因素。在 QuickSort 算法中任务的执行时间可能相差较大,因为 Pivot 的位置依赖于未排序的数据。它们不一定会产生相等大小的数组段(实际上,大小的变化范围很大)。为了弥补这种不均匀性,本公式计算的 depthRemaining 参数值所产生的任务数大于内核数。该公式限制的任务数约为内核数目的 16 倍。之所以如此是因为任务数不能大于 $2^{\text{depthRemaining}}$ 。如果用 $\text{depthRemaining} = \log_2(\text{NCores}) + 4$ 来替换并进一步简化可得任务数目为 $16 \times \text{NCores}$ 。(对于任意的 a , 如果 $a = \log_2(b)$, $2^a = b$, 则 $2^{(a+4)}$ 等于 2^a 的 16 倍。)

通过测量递归深度限制子任务数目是一项非常重要的技术。

6.3 变化形式

动态任务并行存在以下几个变形。

6.3.1 while-not-empty 并行

本章目前涉及的例子采用到的技术属于深度优先遍历的并行类别。对于其他遍历方式也有相应的并行算法。这些技术都依赖于并发集合所处理

的工作。以下是代码示例。

```
public static void ParallelWhileNotEmpty<T>(
    IEnumerable<T> initialValues,
    Action<T, Action<T>> body)
{
    var from = new ConcurrentQueue<T>(initialValues);
    while (!from.IsEmpty)
    {
        var to = new ConcurrentQueue<T>();
        Action<T> addMethod = to.Enqueue;
        Parallel.ForEach(from, v => body(v, addMethod));
        from = to;
    }
}
```

代码显示了如何利用 **Parallel.ForEach** 处理初始集合。在处理过程中，可能会发现其他的值，此时将其放置到队列中。第一批值处理完成后，开始处理其他值，该过程可能产生更多要处理的值。整个过程重复直到不再产生其他值。

ParallelWhileNotEmpty 方法可以用于访问二叉树。

```
static void ParallelWalk4<T>(Tree<T> tree, Action<T> action)
{
    if (tree == null) return;
    ParallelWhileNotEmpty(new[] { tree }, (item, adder) =>
    {
        if (item.Left != null) adder(item.Left);
        if (item.Right != null) adder(item.Right);
        action(item.Data);
    });
}
```

网站链接检测工具是一个恰当使用 `ParallelWhileNotEmpty` 方法的例子。遍历任务加载初始页面并搜索链接，将每个检测到的链接从列表中移除，并将相同站点的其他未检测的链接添加到列表中。最后当不存在未检测的链接时，应用程序终止。

6.3.2 任务链与父子任务

TPL 包含一个名为 `AttachedToParent` 的任务创建选项。该选项常用于动态任务并行模式的代码中，其目的是将子任务链接到任务创建者。此时，被创建的任务称为子任务，创建者则称为父任务。

以下两种情形可以使用 `AttachedToParent` 选项：一种情况是需要将父任务的状态链接到子任务；第二种情况是需要通过 Visual Studio 进行开发并调试和查看父子关系。任务的执行顺序不受该 `AttachedToParent` 选项的影响。

以下是代码示例。

```
static void ParallelWalk2<T>(Tree<T> tree, Action<T> action)
{
    if (tree == null) return;
    var t1 = Task.Factory.StartNew(
        () => action(tree.Data),
        TaskCreationOptions.AttachedToParent);
    var t2 = Task.Factory.StartNew(
        () => ParallelWalk2(tree.Left, action),
        TaskCreationOptions.AttachedToParent);
    var t3 = Task.Factory.StartNew(
        () => ParallelWalk2(tree.Right, action),
        TaskCreationOptions.AttachedToParent);
    Task.WaitAll(t1, t2, t3);
}
```

`AttachedToParent` 选项会影响父任务的行为。如果某个父任务拥有一个或多个运行状态的子任务，当父任务因为某些原因结束运行时，其 `Status` 属性将变为 `WaitingForChildrenToComplete`。只有当所有子任务运行结束后，父任务的 `Status` 属性才会变为图 6.1 所示的三种状态之一。

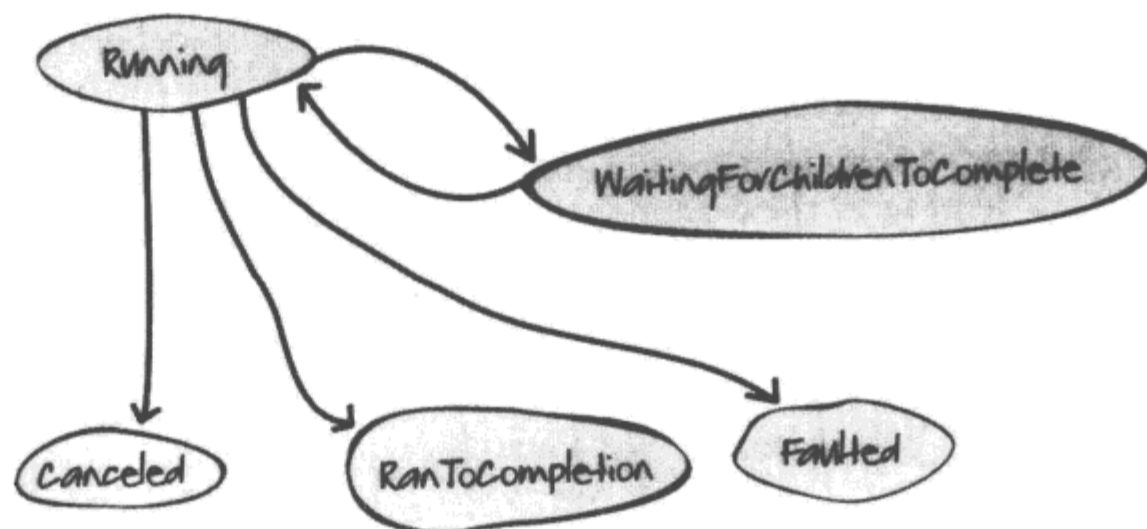


图 6.1 拥有子任务的父任务的生命周期

异常来源于子任务，由父任务进行捕捉。

要访问 Visual Studio 调试器的父/子视图，只需右击“并行任务”窗口的首行，并选择“父子视图” (Parent Child View)。

6.4 设计说明

使用默认任务调度程序时，要注意长时间运行的处理器密集型的任务。

本书第 3 章描述了 Microsoft .NET Framework 的默认调度程序。默认任务调度程序为每个工作线程启用快速本地队列，以及为了负载平衡使用 `work stealing` 算法。这些特征对于采用动态任务并行机制的应用的性能至关重要。

默认任务调度程序的线程注入试探式算法并不适用于所有的动态任务并行机制。尤其对于长期运行、处理器密集型的任务，分析应用可能会显示处理器超额申请(工作线程太多)。如果出现这种情况，则需考虑将问

题分解为更短的任务。例如在并行快速排序中，可以通过调整 `depthRemaining` 参数值来决定创建更多还是更少的任务。

也可以编写一个自定义线程调度程序。更多关于任务调度的知识参见本书第 3 章。

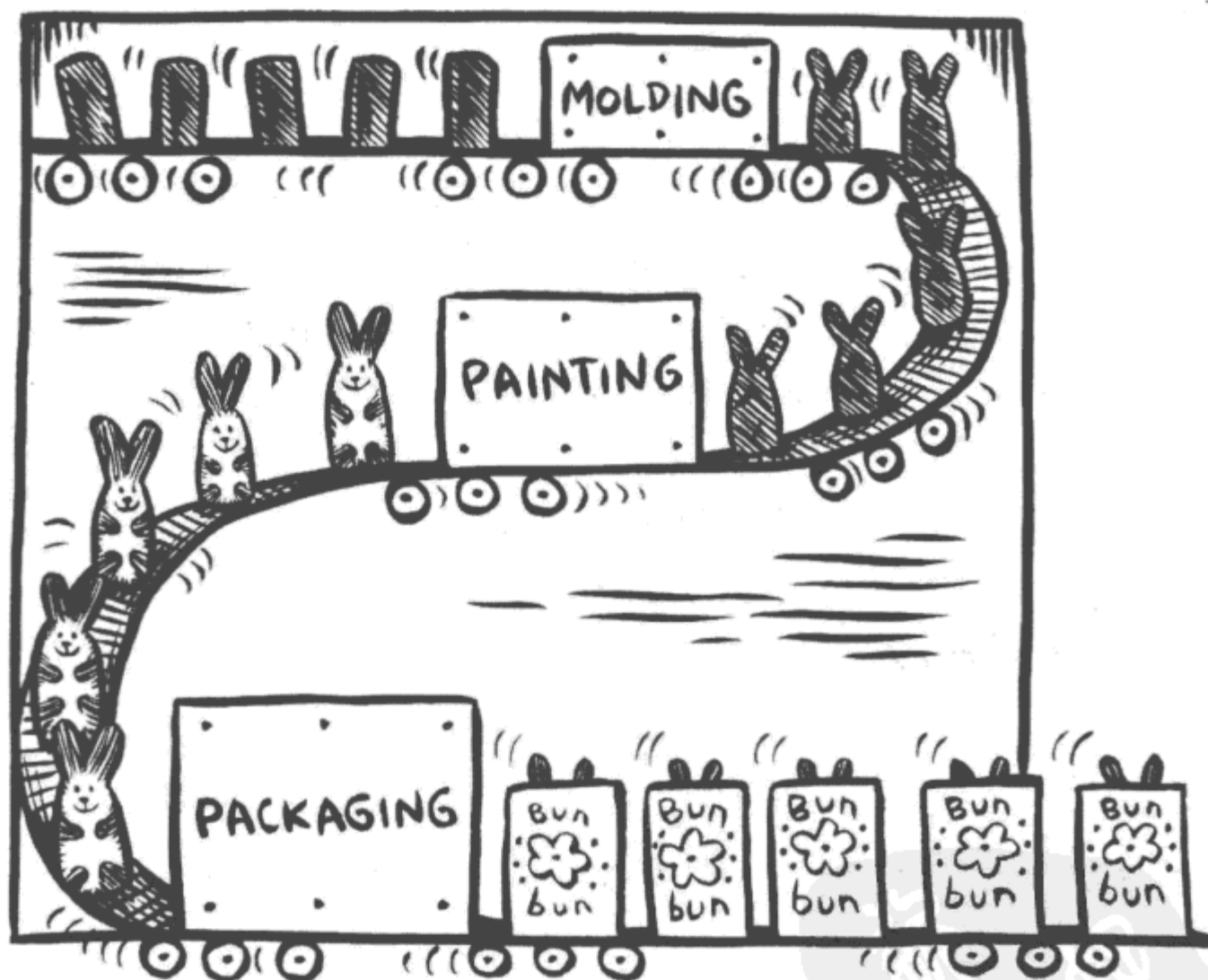
6.5 练习

1. CodePlex 示例代码为 `Threshold` 设置了一个特定的默认长度值。此时 `QuickSort` 转化为非递归的 `InsertionSort` 算法。通过命令行参数可以设置不同的 `Threshold` 值，并观察对于不同大小数组排序所需的执行时间。你会看到什么结果？系统最佳 `Threshold` 值是什么？
2. 通过命令行参数调整数组大小，并观察顺序版本和并行版本函数执行的时间开销。结果是多少？解释观察值。
3. 除了内核数外，提出其他限制任务数目的衡量标准。

6.6 扩展阅读

Toub 讨论了并行 `QuickSort` 算法的其他变化形式。

- Toub S. *Patterns of Parallel Programming: Understanding and Applying Parallel Patterns with the .NET Framework 4*. 2009. <http://www.microsoft.com/downloads/details.aspx?FamilyID=86b3d32b-ad26-4bb8-a3ae-c1637026c3ee&displaylang=en>.



第 7 章 流 水 线

流水线模式利用并行任务以及并发队列处理顺次输入的值。每个任务执行流水线的一个阶段，尽管输入值是按序处理的，队列作为缓冲区允许流水线各个阶段并发执行。可以将软件流水线想像成工厂的生产线。项目的组装一个阶段一个阶段地进行。整个生产线的输入输出顺序完全一致。

流水线由一系列的并行任务和缓冲区连接而成。

流水线由一系列的生产者/消费者阶段组成，每个阶段都依赖于前一阶段的输出。当使用并行循环存在太多依赖时，可以使用流水线进行并行计算。

流水线技术有很多用途。当我们从实时事件流接收数据时，流水线技术非常有用，如股票收报数据、鼠标点击事件和网络数据包到达事件等。流水线技术也常用于处理数据流，如进行压缩加密或处理视频帧。综上所述，数据元素的顺序处理是很重要的。

不要将流水线技术和并行循环混淆。只有在无法应用并行循环的情况下，才采用流水线技术。通过流水线模式，数据可以进行有序的处理。第一个输入转换为第一个输出，第二个输入转换为第二个输出，以此类推。

流水线技术对于集中式应用以及 I/O 操作频繁的应用也十分有用。

7.1 基础

在微软的 .NET Framework 中，连接软件流水线的各个阶段的缓冲区通常都是基于 `BlockingCollection<T>` 类的。

图 7.1 显示了某流水线的四个阶段。第一阶段从数据源读取字符或语句片段，第二阶段修正标点符号和大小写，第三阶段组织成完整的语句，第四阶段将语句写入磁盘。

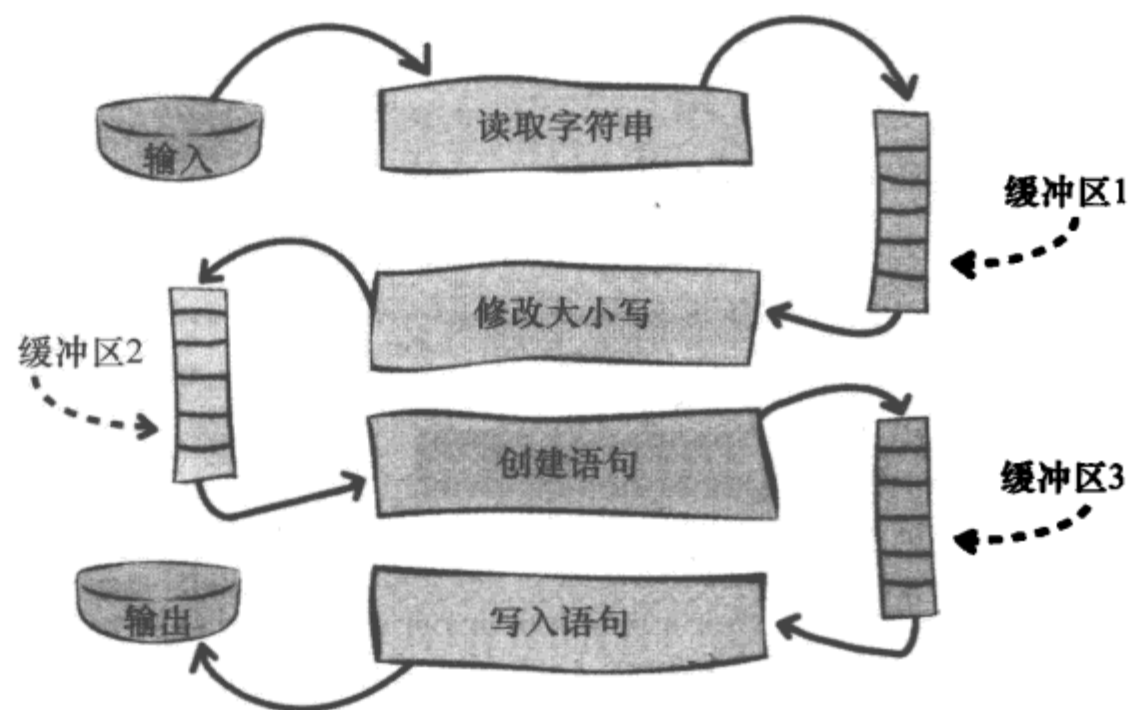


图 7.1 流水线示例

流水线的各个阶段都有特定的输入输出。例如，图中的“读取字符串”任务从源文件获得输入并写入第一个缓冲区。由于并发队列缓冲了所有的共享输入输出，流水线的各个阶段可以同时执行。如果有四个可用的内核，则这些阶段可以并发执行。如果输出缓冲区空闲，流水线就可以产生值并添加到输出队列中。如果输出缓冲区已满，生产者将等待，直到缓冲区可用。流水线的输入阶段也可能等待(即阻塞)。对于编程而言等待输入是再熟悉不过的情形了——如果枚举或流为空，则消费者将等待直到有可用值或者读到“文件结束”标记。当集合阻塞时，其工作方式也是这样的。使用缓冲来保存多个值而不是一个值，弥补了处理每个值所需的时间可能不同的缺陷。

如果队列已满，生产者阻塞；如果队列为空，消费者阻塞。

`BlockingCollection<T>`类通过 `CompleteAdding` 方法标记“文件结束”。该方法通知消费者当前面添加进集合中的所有数据移除或处理之后，结束处理循环。

以下代码演示了如何使用 `BlockingCollection` 类的缓冲区和流水线各个阶段的任务。

```
int seed = ...
int BufferSize = ...
var buffer1 = new BlockingCollection<string>(BufferSize);
var buffer2 = new BlockingCollection<string>(BufferSize);
var buffer3 = new BlockingCollection<string>(BufferSize);

var f = new TaskFactory(TaskCreationOptions.LongRunning,
                        TaskContinuationOptions.None);

var stage1 = f.StartNew(() => ReadStrings(buffer1, ...));
var stage2 = f.StartNew(() => CorrectCase(buffer1, buffer2));
var stage3 = f.StartNew(() => CreateSentences(buffer2, buffer3));
var stage4 = f.StartNew(() => WriteSentences(buffer3));

Task.WaitAll(stage1, stage2, stage3, stage4);
```

第一个阶段产生输入字符串并放入缓冲区 1 中。第二个阶段转换字符串。第三个阶段将字符串与产生的语句相结合。最后一个阶段将修正的语句写入文件。

这些缓冲区是 `BlockingCollection<string>` 类的实例。构造函数的参数指定了缓冲区的最大值。此处设置的缓冲大小为 32。更多关于如何选取阻塞集合的缓冲区大小的知识参见 7.5 节。

正如本例所示，流水线中的任务通常用 `LongRunning` 选项创建。更多信息参见 7.4 节。

流水线的第一个阶段包含一个写入输出缓冲区的顺序循环。

```
static void ReadStrings(BlockingCollection<string> output,
                        int seed)
{
    try
    {
```

如果忘记了 `LongRunning` 选项也不必担心，如果使用默认的任务调度程序，.NET 线程池将会每隔两秒自动添加额外的工作线程。

```
foreach (var phrase in PhraseSource(seed))
{
    Stage1AdditionalWork();
    output.Add(phrase);
}
finally
{
    output.CompleteAdding();
}
```

在流水线中，用顺序循环处理步骤。当流水线的各个阶段不再产生值时，调用 `CompleteAdding` 方法。

顺序循环构成了数值输出缓冲区。这些值来自外部数据源，它通过 `PhraseSource` 方法访问，返回一个单线程的 `IEnumerable<string>` 类实例。生产者通过 `Add` 方法将值添加到阻塞集合中。如果队列已满，该方法可能阻塞。可以用此方法限制流水线中比其他阶段运行得更快的阶段。

`CompleteAdding` 方法通常在 `finally` 模块中调用，因此即使发生异常该方法也会执行。

流水线的中间阶段处理输入缓冲区中的值，并将新产生的值放入输出缓冲区中。以下代码显示了这一机制。

```
void DoStage(BlockingCollection<T> input,
             BlockingCollection<T> output)
{
    try
    {
        foreach (var item in input.GetConsumingEnumerable())
        {
            var result = ...
            output.Add(result);
        }
    }
    finally
    {
    }
```



```
{  
    output.CompleteAdding();  
}  
}
```

可以查看在线源代码了解 `CorrectCase` 和 `CreateSentences` 方法的具体实现，两者构成了流水线的第二、三阶段。它们的结构和本例非常相似。更重要的是，输入阻塞集合的 `GetConsumingEnumerable` 方法返回一个枚举值，该值是消费者可以“接收”的类型。一个生产者可能对应多个消费者。当某个消费者接收一个值时，该值对其他消费者并不可见。

尽管在本例没有体现，但阻塞集合的 `GetConsumingEnumerable` 方法是可以被多个消费者调用的。这使得生产者的值可以分配给多个接收者。如果某个接收者从阻塞集合中获得一个值，其他消费者无法再获得该值。

一个阻塞集合
可以拥有多个消
费者。

流水线的最后一个阶段只从阻塞集合中获取值，但并不产生值，而是将值写入流中。以下是代码。

```
static void WriteSentences(BlockingCollection<string> input)  
{  
    using (StreamWriter outfile =  
        new StreamWriter(PathForPipelineResults))  
    {  
        // ...  
        foreach (var sentence in input.GetConsumingEnumerable())  
        {  
            var printSentence = ...  
            outfile.WriteLine(printSentence);  
        }  
    }  
}
```

在.NET 中之所以很容易写流水线代码，是利用了一些在顺序编程中熟知的技巧，如通过 `IEnumerable<T>` 进行迭代。其中 `BlockingCollection<T>`

类中隐藏了同步的实现。

(为了让本例更为清晰, 代码中省略了错误处理、取消操作以及性能数据收集的一些详细信息。这些信息参见本章后文将谈到的完整的 ImagePipeline 示例。)

7.2 示例

由于本应用要求图像进行顺序处理, 因此本例不能使用并行循环。并行循环不能保证任何特定的处理顺序。

在线示例中包含一个名为 ImagePipeline 的应用程序。该程序接收 JPEG 图像的目录并生成缩略图, 之后再通过一些图像增强滤镜进行处理。处理后的图像按照文件名的字母顺序通过幻灯片进行展示。

7.2.1 顺序图像处理

图像的处理分为四个步骤: 从文件中加载大型彩色图像, 生成带相框的小缩略图, 添加图像噪声产生斑点效果, 最后在幻灯片中显示处理后的图像。图 7.2 显示了该过程。

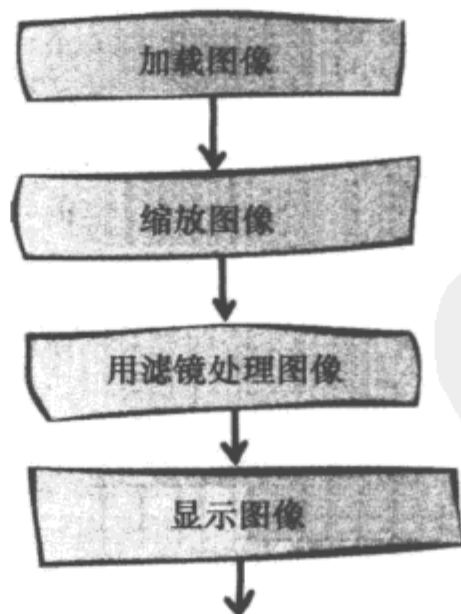


图 7.2 顺序图像处理过程

以下顺序处理的代码。

```
string sourceDir = ...
IEnumerable<string> fileNames = ...
int count = 0;
ImageInfo info = null;

foreach (var fileName in fileNames)
{
    /// ...

    info = LoadImage(fileName, sourceDir, count, ...);
    ScaleImage(info);
    FilterImage(info);
    // ...
    DisplayImage(info, count + 1, displayFn, ...);
    // ...

    count += 1;
    info = null;
}
```

从上面的代码中可以看到，四个步骤分别通过 **LoadImage**、**ScaleImage**、**FilterImage** 以及 **DisplayImage** 四个方法执行。本代码省略了取消操作、错误处理(处理非托管对象)以及性能评估机制。可以参考在线的源代码了解这些细节。更多关于取消操作和错误处理的细节参见 7.3 节。

7.2.2 图像流水线

顺序循环每次只能处理一副图像，每幅图像必须完成所有四个步骤后才能开始处理下一幅图像。实际上，本例似乎有着固有的顺序性——最顶层的循环严格要求图像顺序显示，并且每个子步骤的输入来自前一个子步骤的输出。图像显示不会发生在滤镜处理之前，滤镜处理不会发生在图像缩放之前，而图像缩放也不会发生在图像加载之前。

然而，流水线模式可以将并行机制引入本例中。每幅图像依然顺序执行四个步骤，但不同图像的各个阶段可以并行执行。图 7.3 显示了处理图像的流水线。

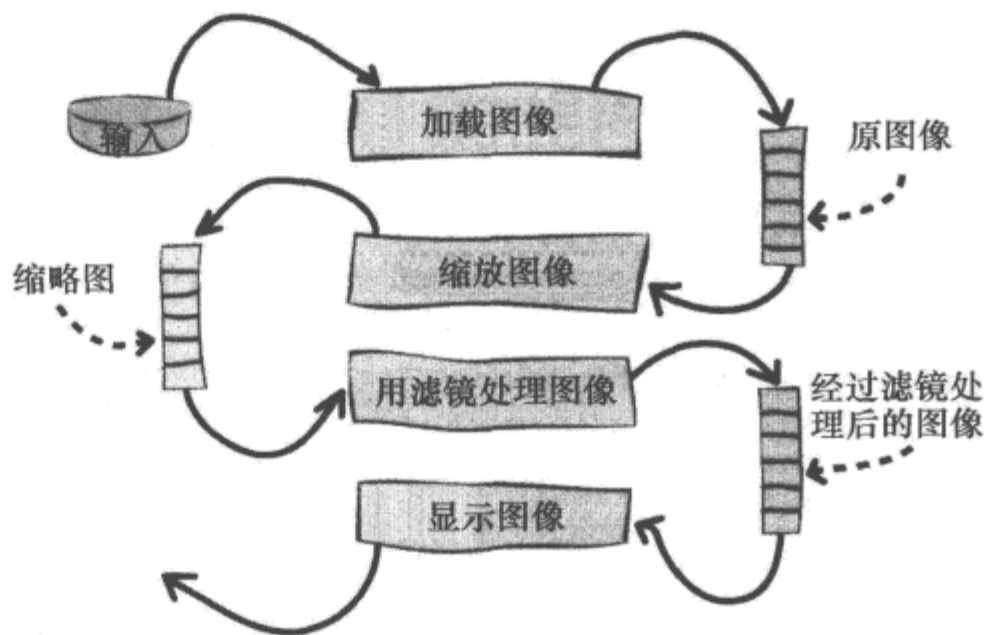


图 7.3 图像流水线

以下为引入并行处理的代码。

```
IEnumerable<string> fileNames = ...
string sourceDir = ...
Action<ImageInfo> displayFn = ...
int limit = ...

var originalImages = new BlockingCollection<ImageInfo> (limit);
var thumbnailImages = new BlockingCollection<ImageInfo> (limit);
var filteredImages = new BlockingCollection<ImageInfo> (limit);
try
{
    var f = new TaskFactory(TaskCreationOptions.LongRunning,
                           TaskContinuationOptions.None);

    // ...

    var loadTask = f.StartNew(() =>
```

```
        LoadPipelinedImages(fileName, sourceDir,
                               originalImages, ...));

    var scaleTask = f.StartNew(() =>
        ScalePipelinedImages(originalImages,
                               thumbnailImages, ...));

    var flterTask = f.StartNew(() =>
        FilterPipelinedImages(thumbnailImages,
                               filteredImages, ...));

    var displayTask = f.StartNew(() =>
        DisplayPipelinedImages(
            filteredImages.GetConsumingEnumerable(),
            displayFn, ...));

    Task.WaitAll(loadTask, scaleTask, flterTask, displayTask);
}
finally
{
    // ... release handles to unmanaged resources ...
}
```

(为了让本例更为清晰，代码中省略了错误处理、取消操作以及性能数据收集的细节。完整的实现参见在线示例。)

在流水线的各个阶段之间有三个阻塞集合扮演着缓冲区的角色。在顺序执行的版本中，四个阶段是相同的。调用任务工厂的 `StartNew` 方法可以将每个处理阶段作为一个长期的任务执行。

调用 `Task.WaitAll` 方法延迟清理工作直到执行完所有的图像处理阶段。

7.2.3 运行特性

通过图 7.4 所示的调度图，我们可以了解顺序流水线的运行特性。

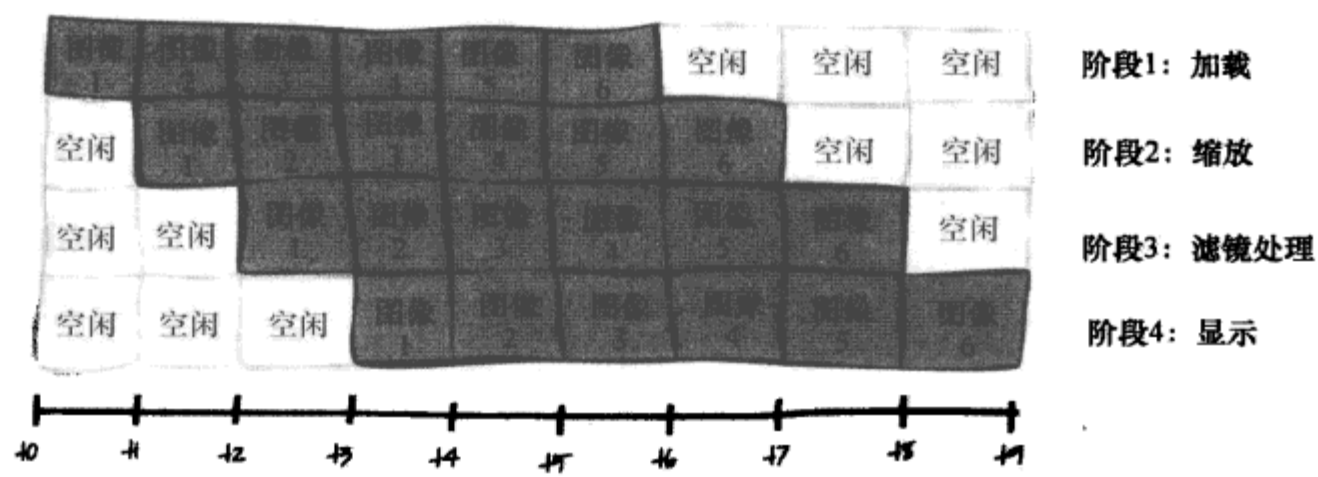


图 7.4 各阶段运行速度相等的图像流水线

图 7.4 显示了一段时间内图像流水线执行的情况。例如，图中的最上面一行表示第一幅图像的处理开始于 t_0 时刻，第二幅图像开始于 t_1 时刻。假设某一时段流水线刚好达到平衡，即流水线每个阶段的工作时间相同，我们将这一时段称之为持续时间 T 。从而图中的 t_1 发生在 T 个时间单位后，而 t_2 则发生在 $2 \times T$ 个单位后，依此类推。

如果有足够的内核允许流水线任务并行执行。图中显示的 6 幅图的 4 个阶段处理用时大约为 $9 \times T$ 。相比之下，顺序处理需要用时 $24 \times T$ ，这是因为顺序处理的 24 个步骤需要逐步执行。

随着处理的图像的数量增加，其平均性能也会提高。这是因为，如图 7.4 所示，启动时，流水线需要依次填充每个阶段，而停止时，流水线需要依次空出每个阶段，这时都有一些闲置的内核没有被利用。但是当有大量的图像时，启动和停止几乎可以忽略，因而每幅图像的平均时间将会接近于 T 。

然而需要注意的是：流水线的各个阶段消耗的时间并不总是相等。图 7.5 的调度图显示了滤镜处理阶段的时间开销是其他阶段两倍的情况。

如果有足够的可用内核且流水线各个阶段的耗时相等，则流水线整体的时间开销将和一个阶段的时间开销相等。

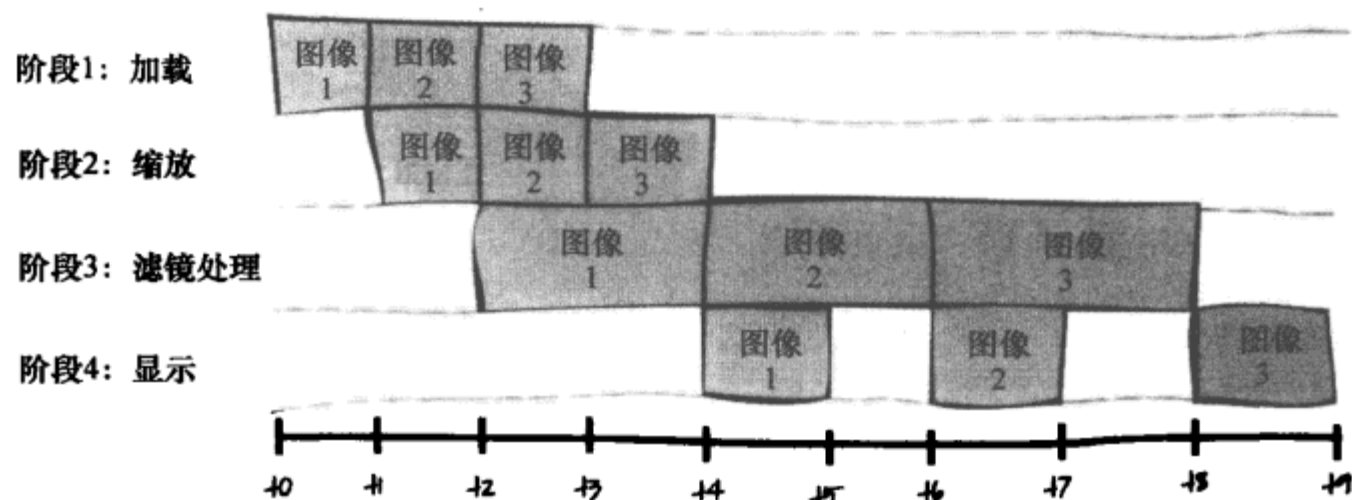


图 7.5 各阶段处理时间不相等的图像流水线

当某个阶段的用时为 $2 \times T$ 个单位时间, 而其他阶段用时 T 个单位时间时, 你会发现不可能保持所有内核都运行。通常而言(对于大量的图像), 处理一幅图像的耗时为 $2 \times T$ 。换句话说, 当可以为流水线的每个阶段分配足够的内核时, 流水线的整体速度将接近最慢阶段的速度。

当流水线各阶段的时间不相等时, 流水线的速度近似等于最慢阶段的速度。

如果在 Visual Studio 编译器中执行 ImagePipeline 应用, 将看到这种影响。ImagePipeline 提供了一个用户界面, 用来报告流水线各阶段所需要的平均毫秒数以及处理每幅图像的总体时间长度的平均值。在顺序模型下(选择 Sequential 按钮), 执行该应用, 每幅图像在稳定状态下的时间开销等于所有阶段的时间和。而在流水线模型下, 时间开销则与最慢的阶段相等。当各个阶段的速度相等时, 流水线的效率最高。虽然不能总是使各阶段速度相等, 但有必要多进行尝试。

7.3 变化形式

流水线模式有多个变形。

7.3.1 取消流水线

流水线的各个任务携手完成工作, 响应取消时它们也必须相互协调。

在第 3 章介绍的标准取消方案中,从应用顶层(如用户界面)传递一个 `CancellationToken` 标记给任务。在流水线的阶段中,需要在两处观察该标记。代码如下。

```
void DoStage(BlockingCollection<T> input,
             BlockingCollection<T> output,
             CancellationToken token)
{
    try
    {
        foreach (var item in input.GetConsumingEnumerable())
        {
            if (token.IsCancellationRequested) break;
            var result = ...
            output.Add(result, token);
        }
    }
    catch (OperationCanceledException) { }
    finally
    {
        output.CompleteAdding();
    }
}
```

第一个需要检查取消标记的地方自然是循环的起点。此处,只需要打断循环,代码如下。

```
if (token.IsCancellationRequested) break;
```

第二个位置则不太明显,利用 `BlockingCollection<T>` 类的 `Add` 重载方法将取消标记作为参数接收,代码如下。

```
output.Add(result, token);
```

如果给 `Add` 方法传递一个取消标记,为了获得新值,阻塞集合将会

一直监听取消请求。

为什么要添加如此细致的取消逻辑呢？要理解这一点，我们需要稍作回顾。流水线的各阶段在调用输出队列的 `Add` 方法创建值时，可能会阻塞。(如果输出队列已满，`Add` 方法的调用要等待空间可用时才返回结果。) 如果此时某个阶段请求取消，而你并没有检测到，就有可能引起死锁。因为流水线各阶段所需要的值可能被当前阶段取消。因此，流水线后面的阶段很可能在未被排出队列的情况下中断。生产者在等待可用空间时被阻塞，因而不能继续向前处理。

错误的取消一个管道可能会引起程序的死锁。要小心遵循本章的指导。

本例中采用的方案是，利用重载的 `Add` 方法接收取消标记。如果生产者在等待可用空间时产生取消请求，阻塞集合将抛出 `OperationCanceledException` 异常。不论循环是正常退出，还是从 `break` 关键字退出或是在取消发生异常的过程中退出，`finally` 语句将确保对所有的输出缓存进行标记。这将使得任何消费者都可能等待输入，并且随时处理取消请求。

在流水线阶段主循环的开始处以及往阻塞集合中添加值时检查取消请求。

尽管阻塞集合等待输入时，也可以检查取消标记(重载的 `GetConsumingEnumerable` 方法接收取消标记)。利用本章描述的技术，则不必这么做。

注意，如果类型 `T` 实现了 `IDisposable` 接口，则在 .NET 的代码规范下，还必须调用 `Dispose` 方法进行取消。你需要对当前迭代对象和存储在阻塞队列中的 `T` 实例进行处理。在线的 `ImagePipeline` 示例展示了完整的做法。

7.3.2 处理流水线异常

异常和取消相似，两者的差别在于，当流水线的某个阶段产生未处理的异常时，其他阶段的任务默认情况下并不会收到通知。在这种情况下，应用程序有几种可能会产生死锁。

当流水线的某个阶段产生未处理的异常时，应当取消其他阶段，否则将会发生死锁。请仔细参照本章的指导。

当异常发生时，利用 `CancellationTokenSource` 的特定实例，可以协调流水线的各个阶段进行关闭。以下是一个例子。

```
static void DoPipeline(CancellationToken token)
{
    using (CancellationTokenSource cts =
        CancellationTokenSource.CreateLinkedTokenSource(token))
    {
        var f = new TaskFactory(TaskCreationOptions.LongRunning,
                                TaskContinuationOptions.None);

        var stage1 = f.StartNew(() => DoStage1(..., cts));
        var stage2 = f.StartNew(() => DoStage2(..., cts));
        var stage3 = f.StartNew(() => DoStage3(..., cts));
        var stage4 = f.StartNew(() => DoStage4(..., cts));
    }
    Task.WaitAll(stage1, stage2, stage3, stage4);
}
```

`CancellationTokenSource` 类的 `CreateLinkedTokenSource` 方法可以创建一个句柄来响应一个外部取消请求，也可以初始化(确认)自身的外部取消请求。该方法接收取消标记源作为参数，执行流水线的各个阶段，因此每个阶段可以用标记源初始化取消标记。

以下是一个例子。

```
void DoStage(BlockingCollection<T> input,
             BlockingCollection<T> output,
             CancellationTokenSource cts)
{
    try
    {
        var token = cts.Token;
        foreach (var item in input.GetConsumingEnumerable())
```

```
{
    if (token.IsCancellationRequested) break;
    var result = ...
    output.Add(result, token);
}
}
catch (Exception e)
{
    // If an exception occurs, notify all other pipeline stages.
    cts.Cancel();
    if (!(e is OperationCanceledException))
        throw;
}
finally
{
    output.CompleteAdding();
}
}
```

这段代码和前面描述的取消形式相似。唯一的不同在于，当产生未处理的异常时，`catch` 模块将捕捉该异常，并向流水线的其他阶段发送取消信号，从而使得流水线的各个阶段顺序关闭。

当所有的流水线任务停止之后，作为 `AggregateException` 内部异常的实例，`Task.WaitAll` 方法将抛出原始异常。应当使用 `catch` 或 `finally` 模块确保清理工作，如释放非托管资源的句柄。

你可能会疑惑为什么不直接传递 `CancellationTokenSource` 对象到 `DoPipeline` 方法而选择传递 `CancellationToken` 值。按照规范，只有取消标记可以作为参数传递。`CancellationToken` 的传递允许低版本的类库代码响应外部取消但不允许用低版本的类库初始化高版本的取消组件。换句话说，响应取消请求比初始化取消请求所需的权限低。

7.3.3 利用多个生产者实现负载均衡

`BlockingCollection<T>` 类允许从多个生产者读取值。该功能由 `TakeFromAny` 静态方法及其变量提供。对于一些流水线应用(并非全部), 可以通过 `TakeFromAny` 实现负载均衡策略。这种形式有时也被称为非线性流水线。

本章前面描述的图像流水线的例子要求幻灯片的缩略图演示顺序和文件输入顺序一致。对于很多流水线应用场景, 如视频帧的处理, 这都是通用的约束。然而在图像流水线例子中, 连续图像的滤镜处理操作是相互独立的。在这种情况下, 可以插入额外的流水线任务。其过程如图 7.6 所示。

有时候我们会通过增加特定流水线阶段的任务数量来实现负载均衡。

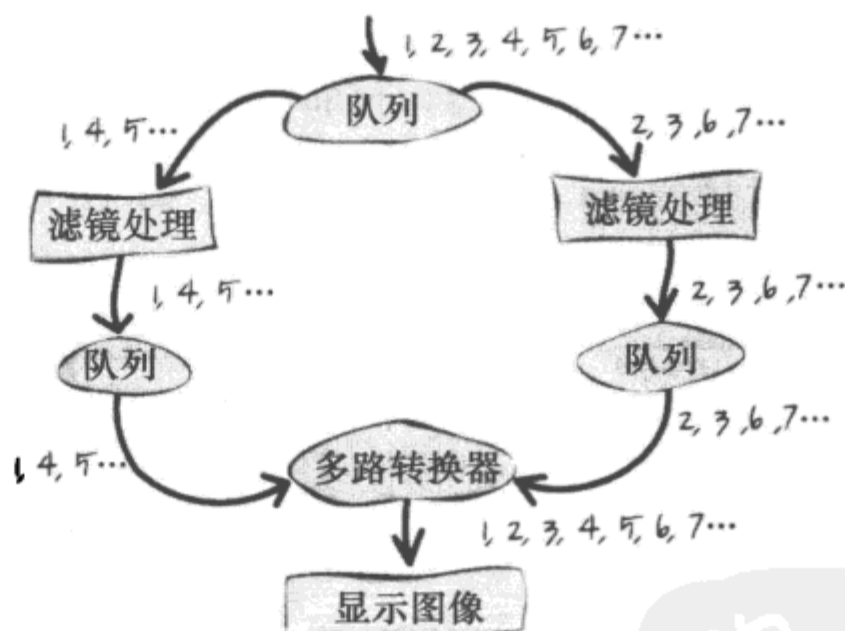


图 7.6 从多个生产者中消费值

图 7.6 显示添加额外的滤镜处理任务的情况。这些任务都接收前一个流水线阶段产生的图像。尽管从滤镜的角度来看, 任何输入图像都是顺序到达的, 但消费这些图像的顺序并不完全确定。

每个滤镜处理阶段有一个阻塞集合用于存放产生的队列元素。而这些队列的消费者是一个多路转换器组件, 其输入连接着所有的生产者。多路

转换器允许消费者在流水线的显示阶段按正确的顺序接收图像。图像并不需要分类或重新排序。取而代之的是，每个生产者队列局部有序，它允许多路转换器通过同时监测所有生产者队列的头部来查找序列的下一个值。这也是阻塞集合 `TakeFromAny` 方法的作用。该方法允许多路转换器阻塞直到任何生产者队列有可读的值为止。

接下来是更为具体的例子。假定每幅图像拥有一个唯一序列号属性，并且图像序号是连续增长的。如图 7.6 所示，第一个滤镜处理序号为 1、4、5 的图像，第二个滤镜处理序号为 2、3、6、7 的图像。每个负载平衡的滤镜收集输出图像并放入两个队列中。两个输出队列都是正确排序的(即低序号的图像之前没有更高的序号的图像)。但是在序号之间会有空缺。举例来说，从第一个滤镜的输出队列中我们可以获得序号为 1 的图像，接下来序号为 4，最后序号为 5，而序号为 2、3 的图像缺失。因为它们在第二个滤镜的输出队列中。

这种缺失是一个需要考虑的问题。下一个流水线阶段，即显示图像阶段，需要依次完整地显示图像(没有缺失值)。这正是多路转换器派上用场的地方。通过 `TakeFromAny` 方法，多路转换器等待来自两个滤镜处理阶段生产者队列的输入。当图像到达时，多路转换器检查图像的序号是否是预期序列的下一个序号，如果是，则将其传送至显示图像阶段；如果不是，多路转换器在内部的前瞻缓冲区中保留该值，并接收另一个输入队列中的值，重复此操作。该算法允许多路转换器把生产者队列的所有值放到一起，确保其有序性而无需对值进行排序。

图 7.7 表明滤镜处理阶段采用双滤镜方式时，其性能是其他流水线阶段的两倍。

除了滤镜处理阶段以外，如果所有流水线阶段处理一幅图像需要 T 个单位时间，而滤镜处理阶段需要 $2 \times T$ 个单位时间。则用两个滤镜和两个生产者队列负载平衡流水线，可以使得处理每幅图像的整体速度近似为

T。在运行 ImagePipeline 例子时选择 Load Balanced 单选按钮，可以看到这种效果。流水线的速度(在处理完适当序号的图像之后)将综合最慢的实例阶段的平均时间以及滤镜处理时间开销的一半，选择两者中的较高者。

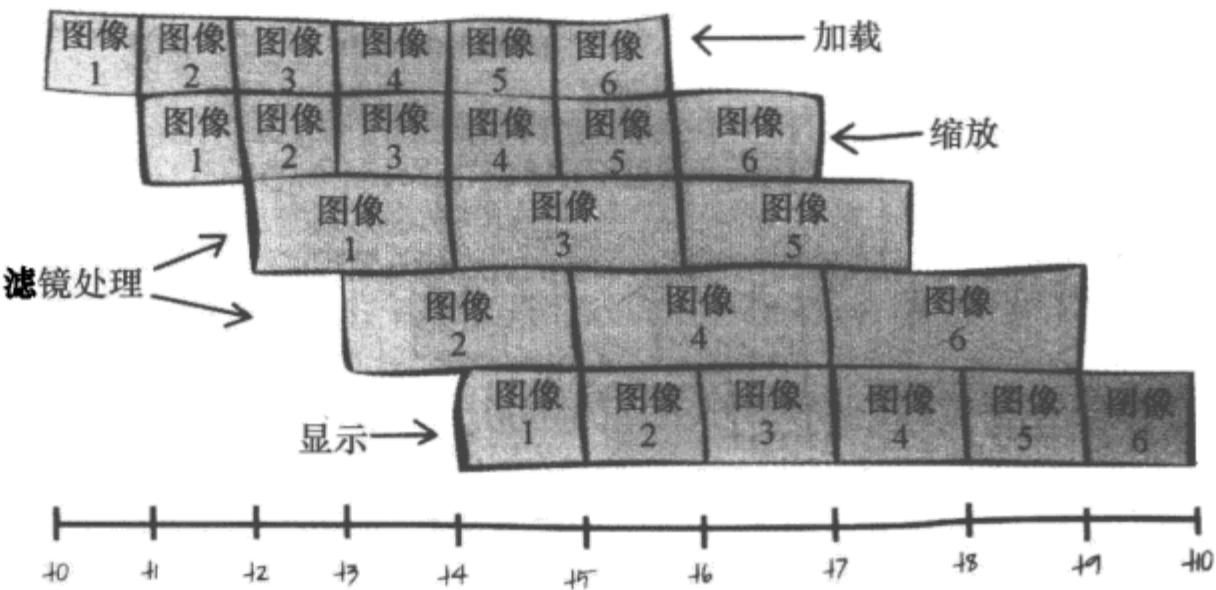


图 7.7 负载均衡的图像流水线

队列 3 的等待时间表明了等待多个生产者队列引入的开销，ImagePipeline 示例的用户界面对其进行了展示。这个例子说明了在能够提高内核利用率的情况下，如何通过增加并行计算的开销来提高整体的速度。

7.3.4 流水线和流

你可能已经注意到阻塞集合和流具有一些相似性。有时把阻塞集合当作流是有效的，反之亦然。举个例子，你可能想通过流水线模式的类库方法对流进行读和写的操作。假如你想对某个文件进行压缩和加密。NET Framework 提供了这两种操作的支持，但是不能以流作为这些方法的输入，而阻塞集合则可以。

可以创建一个底层实现依赖于任务和阻塞集合的流。更多相关信息参见 7.8 节。

7.3.5 异步流水线

到目前为止，我们介绍的都是同步流水线。生产者和消费者长期运行输入和输出阻塞的任务。也可以拥有一个异步的流水线，直到数据可用时才创建任务。要做到这一点，可以利用 `ParallelExtensionsExtras` 示例项目中的 `AsyncCall` 类。(该示例的更多信息参见 7.8 节。) `AsyncCall` 类是一个队列，生产者将产生的数据放入其中。当数据到达时，如果当前没有处理队列的任务，则会激活产生一个新的处理任务。如果没有更多数据，则任务将停止。如果有更多数据到达，将启动新任务。

7.4 反模式

实现流水线时，需要注意以下几点。

7.4.1 线程饥饿

流水线要求所有的任务并发执行。如果没有足够的线程运行所有的流水线任务，阻塞集合可能会填满和阻塞。此时，第 3 章介绍的任务嵌套也不起作用。为了确保执行每个流水线任务时，线程可用，可以采用默认任务调度程序的 `LongRunning` 任务创建选项。如果忘记了该选项而使用默认任务调度程序，这仍然安全。最终线程会通知前面的流水线阶段不再执行，并且为后面的流水线阶段添加额外的线程。

7.4.2 阻塞集合无穷等待

如果流水线任务抛出异常，它将不再接受来自阻塞集合的输入。如果阻塞集合已满，写入操作将会阻塞。可以通过 7.3.1 节介绍的“取消流水线”技术避免这种问题。

7.4.3 忘记 GetConsumingEnumerable()方法

由于 `BlockingCollection` 类实现了 `IEnumerable<T>` 接口，所以很容易忘记调用 `GetConsumingEnumerable` 方法。注意，列举阻塞集合的实例不会消耗值。

阻塞集合实现了 `IEnumerable<T>` 接口，因此我们很容易忘记调用接口的 `GetConsumingEnumerable` 方法。如果出现该问题，枚举将获得阻塞集合状态的快照，并且在任何情况下，枚举结果不会从集合去除或修改。这意味着可能会有多个消费者获得相同的值。

7.4.4 采用其他生产者/消费者集合

`BlockingCollection<T>` 类采用并发队列作为默认存储机制。你也可以自己指定存储机制。唯一的约束是底层存储机制必须实现 `IProducerConsumerCollection` 接口。采用 .NET Framework 提供的阻塞集合通常比自己写的实现更简单和安全。

.NET Framework 提供了几种 `IProducerConsumerCollection` 接口实现。其中包括 `ConcurrentBag` 和 `ConcurrentStack` 类。因此，原则上流水线之间的缓冲区可以采用包(无序)或栈[后进先出(LIFO)]机制。

通常只推荐默认的先进先出(FIFO)顺序。如果采用并发的包机制，流水线各个阶段将不依赖顺序输出。在这种情况下，流水线可以用并行循环替代。并行循环的代码既快捷又简单。事实很明显，采用无序的缓冲区是一种错误的模式。

7.5 设计说明

采用流水线分解问题时，需要考虑为流水线设计多少个阶段，这取决于运行时可用的内核数。和本书介绍的其他模式不同，流水线模式不会随着内核数目自动缩放。这是流水线模式的局限性，除非为流水线引入额外的并行机制。

流水线设计的阶段越多，工作得越好，除非缓冲区有过多的移除和添加操作。流水线的某个阶段只执行少量工作也是一个很常见的问题。

为了提高并行度，需要注意使流水线各个阶段执行的时间相近。如果不这么做，流水线的效率将受到性能最低的组件的制约并且导致处理器的申请不足。

缓冲区的大小设置对于整体性能很重要。如果缓冲区设置得太小，尤其是接收的缓冲元素处理时间各异时，将会引起流水线阻塞。在处理过程中，为了协调变量要分配较大的缓冲区。缓冲区的大小还依赖于它所包含的对象的大小。如果缓冲区元素所包含的对象是占用较大内存的位图，你可能希望缓冲区中的项尽量少。

通常，流水线的缓冲区大小应当足以接收变量，但也不能太大。可以通过分析器了解流水线的字符吞吐量，然后修改缓冲区大小使得流水线各个阶段 I/O 阻塞时间最少。

7.6 相关模式

流水线模式与操作系统的管道以及滤镜概念有许多相似之处。

流水线模式是消费者/生产者模型常用技术方案。流水线由一系列的生产者/消费者构成，每个阶段都依赖于前一个阶段的输出。

7.7 练习

1. 修改 7.1 节的例子，编写自己的流水线。
2. 用 Concurrency Visualizer 执行代码，查看并解释结果。

7.8 扩展阅读

Toub 讨论了上层任务以及阻塞集合的流的实现, 将其称为“转移流”。Campbell 讨论了输入来自多个生产者队列的多路转换。Buschmann 讨论了通过操作系统的 shell 命令使用管道和筛选器。MSDN 提供的并行实例包是 ParExtSamples。

- Buschmann F, Meunier R, Rohnert H, Sommerlad P, Stal M. *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*. Wiley, 1996.
- Campbell C, Veanes M, Huo J, Petrenko A. Multiplexing of Partially Ordered Events. TestCom 2005. Springer Verlag, June 2005. <http://research.microsoft.com/apps/pubs/default.aspx?id=77808>.
- Toub S. Patterns of Parallel Programming: Understanding and Applying Parallel Patterns with the .NET Framework 4. 2009. <http://www.microsoft.com/downloads/details.aspx?FamilyID=86b3d32b-ad26-4bb8-a3ae-c1637026c3ee&displaylang=en>.
- ParExtSamples 软件. Samples for Parallel Programming with the .NET Framework 4. <http://code.msdn.microsoft.com/ParExtSamples>.



附录 A 改写面向对象模式

外观模式、装饰模式和库都是面向对象的固定模式，这些模式可能已经成为你的开发工具中的一部分。本附录将介绍如何改写上述模式及其他模式，以便将其应用于并行编程。通过应用某些上述模式，可以降低并行代码的复杂度。而且你也将学到如何避免一些代码陷阱。本附录并没有完整地列出所有的面向对象模式。它着重介绍了一些相当常用的模式，并讨论了在并行编程中使用它们时需要考虑的问题。

如果对面向对象模式不甚了解，可阅读 A.8 节中推荐的参考资料。

A.1 结构模式

外观模式、适配器模式和装饰模式都是结构模式，它们可以隐藏并行的复杂性于一个简单的接口之后。当然，任何时候当增加另一层抽象层，简单性和性能之间总会存在一个折中。在使用这些模式之前必须要考虑这个折中。

使用任何结构模式来隐藏底层的并行性时，其方法和属性都应该复制值或者暴露不可变类型，而不是提供包含可变状态的共享对象的引用。使用不可变类型或者复制对象可以移除并行任务间的依赖性，同时也阻止了并行任务间相同对象(引用)共享的可能性。如果需要对编写调用代码的开发者隐藏并行性，那么这点尤为重要。开发者可能不会意识到传递共享引用或者可变对象对将要并行执行的代码会带来什么问题。为使调用者免受这些问题的干扰，代码实现可能需要额外的类型映射编码。这个方法的详情参见 A.4 节。

A.1.1 外观模式

外观模式提供了一个庞大系统的简化视图。你可能想要应用这个模式或者应用使用了此模式的库来隐藏应用程序中其他部分的并行复杂度。你可以使用一个外观模式来隐藏并行功能于某个串行 API 之后，该 API 可以保护开发者免受并行化造成的任何负面影响。

外观模式会增加应用程序的结构复杂度。只有在基于外观模式的系统本身已经很复杂的情况下，外观模式才会使事情变得简单一些。不过，如果正添加一个外观来隐藏并行化，那么无论系统是否复杂，都可能需要用到它。

1. 示例

在第 5 章中展示的有关应用程序的例子中，对于底层的应用程序模型，视图模型充当着外观模式的角色。视图模型抽象化了应用程序中用户界面层的内部工作。该视图对视图模型的内部工作是完全不知情的。相反，它将被动地依赖视图模型来设置自身的属性。更多信息参见 A.8.3 节。

2. 指南

以下是一些使用外观模式的指南。

- 外观模式通过隐藏并行实现的细节于一个简单的接口之后来降低复杂度。
- 外观模式可以很容易地进行重构，基于可用的硬件或者问题规模，可以分别使用特殊子系统的各种不同(并行化)实现，而不需要改变所调用的代码。
- 前面的两个观点都是外观模式的一般属性。在处理并行化的复杂度时，它们都是相当有用的。

A.1.2 装饰模式

装饰模式重载了底层类的行为。它使用了“包含”关系和继承。可以装饰串行化实现来创建并行化实现。正如外观模式一样，装饰模式可以隐藏并行化的复杂度于调用代码之后。

装饰模式的一个看似明显的示例是为非线程安全的类型写一个锁装饰器。在此有两个理由来说明为什么要避免这个方法。一个锁装饰模式未必会有好的并行性能，除非锁没有被连续地使用。而且，当把锁添加到个别方法时，看起来像是要增加线程的安全性，但这些方法的结合可能仍然需要额外的锁来保证其正确性。

1. 示例

接下来的例子展示了一个装饰的 `IImageEditor` 接口。底层的 `SerialEditor` 实现被 `ParallelEditor` 类装饰，该类是一个调用装饰类型的并行化实现。

```
public interface IImageEditor
{
    void Rotate(RotateFlipType rotation,
               IEnumerable<Bitmap> images);
}
public class SerialEditor : IImageEditor
{
    public void Rotate(RotateFlipType rotation,
                     IEnumerable<Bitmap> images)
    {
        foreach (Bitmap b in images)
            b.RotateFlip(rotation);
    }
}
public class ParallelEditor : IImageEditor
```

```
{  
    private IImageEditor decorated;  
  
    public ParallelEditor(IImageEditor decorated)  
    {  
        this.decorated = decorated;  
    }  
  
    // 修改过的行为  
    public void Rotate(RotateFlipType rotation,  
                      IEnumerable<Bitmap> images)  
    {  
        if (decorated == null)  
            return;  
        Parallel.ForEach(images, b =>  
        {  
            b.RotateFlip(rotation);  
        });  
    }  
    // 额外的行为  
}
```

这个例子创建了一个装饰的 **ParallelEditor** 对象，并传递了一个 **SerialEditor** 的实例给构造函数。**ParallelEditor** 修改了 **Rotate** 方法中的行为，使其运行 **Parallel.ForEach** 循环，但它并没有取代 **SerialEditor** 类中的任何方法，如以下代码所示。

```
IList<Bitmap> images = new List<Bitmap>();  
//加载图像  
  
IImageEditor parallel = new ParallelEditor(new SerialEditor());  
parallel.Rotate(RotateFlipType.RotateNoneFlipX, images);
```

装饰的 `Rotate` 方法不能访问可变的共享状态，这点非常重要。
`Parallel.ForEach` 的每个步骤必须是独立的。

2. 指南

以下是使用装饰模式的一些指南。

- 装饰模式的重要优势是它在保留现存接口的同时也封装了并行性。调用的代码大部分保留不变。
- 装饰模式允许基于可用的硬件数量或问题规模使用不同的并行实现，不用改变调用代码。
- 装饰模式将关注点分离开来。装饰类型执行正要做的工作，而装饰器只负责工作的并行化。
- 前两个观点是装饰模式的一般属性。在处理并行性复杂度的时候，它们会相当有用。

A.1.3 适配器模式

适配器模式将一个接口转换成另一个。这个模式对并行化很有用，因为它跟外观模式一样，都允许隐藏复杂度，并暴露出便于开发者使用的接口。

1. 示例

接下来的代码暴露了一个基于事件的名为 `IWithEvents` 的接口，和一个使用 `future` 的名为 `IWithFuture` 的接口。本段代码的目标是使用对开发者更为熟悉的接口，因为有些开发者不是并行编程的专家或者他们更喜欢基于事件的模型。

```
public interface IWithFutures
{
    Task<int> Start();
}
```

```
}

public class FuturesBased : IWithFutures
{
    public Task<int> Start()
    {
        return Task<int>.Factory.StartNew(() =>
        {
            // ...
        });
    }
}

public interface IWithEvents
{
    void Start();
    event EventHandler<CompletedEventArgs> Completed;
}

public class EventBased : IWithEvents
{
    readonly IWithFutures instance = new FuturesBased();

    public void Start()
    {
        Task<int> task = instance.Start();
        task.ContinueWith((t) =>
        {
            var evt = Completed;
            if (evt != null)
                evt(this, new CompletedEventArgs(t.Result));
        });
    }
}
```



```
public event EventHandler<CompletedEventArgs> Completed;
}
```

基于事件的实现改写了 `IWithFutures` 接口，并允许其结果由一个事件处理程序来处理。以下的代码就展示了这个实现。

```
IWithEvents model = new EventBased();
bool completed = false;

model.Completed += (s, e) =>
{
    Console.WriteLine("Completed Event: Result = {0}", e.Result);
    completed = true;
};

//模型开始并等待事件的完成
model.Start();
```

2. 指南

使用适配器模式来修改现存的接口对于那些对并行化的使用不是很熟悉的开发者而言会更容易。

A.1.4 库和并行数据访问

库调解了应用程序的逻辑和数据访问层之间的矛盾。可以想象，库对于一个或更多的数据源来说，其角色相当于一个外观模式的应用程序。在 .NET Framework 的库中通常使用 ADO.NET 来访问数据库。本节内容讨论了从某个并行应用程序中调用基于 ADO.NET 的库，而不是直接实现库本身时，应该考虑什么。

ADO.NET 用于优化吞吐量和扩展性。ADO.NET 对象不会锁定资源，而且它也只能在单个线程上使用。ADO.NET 实体框架(EF)建立在 ADO.NET 之上，它们之间拥有相同的约束。如果考虑到这些约束条件，

应用程序该如何从多任务中访问库？以及何时才适合访问？

从并行任务中进行数据访问与从 ASP.NET 的线程池中访问数据是相似的。ASP.NET 和数据访问指南所应用的技术大部分是相同的。尤其是，当从多任务中连接一个数据库时，它们是不会在任务中共享该连接的，而且该连接不会长时间持续打开着，以便于该连接可以被再次使用。

将多个并行任务连接到单个数据库看起来可能会是提高性能的好方法。例如，你可能会好奇为什么没有“PLINQ-to-SQL”这样的方法。事实上，数据访问中使用多任务可以减少单个用户访问数据库的时间，但相应地它会影响整个数据库的吞吐量。换句话说，单个用户将会单独使用很多数据库资源，这样的话可以使得他们自身的运行速度增快，但这是以限制访问数据库的用户个数为代价而实现的。

对于要融合从多个不同数据源中查询得到的结果的应用程序来说，并行化是一个合适的方法。运行在 Windows Azure 上的应用程序是很好的例子。Windows Azure 应用程序通常将数据存储在一个存储表和存储块的混合体之中，而且可能会因为扩展性和性能要求将数据分散在多个数据库中。

1. 示例

关于如何并行连接多个不同数据库的例子请参见 A.8 节推荐的参考资料。

2. 指南

以下有一些关于数据访问的指南。

- 不要在任务之间共享 ADO.NET 的连接。ADO.NET 不是线程安全的。每个任务应该使用自己的连接。
- 保持数据源连接为打开状态的时间要尽量短。明确地关闭连接来确保没被用到的连接不会保持打开状态。不要期待垃圾回收站会帮你处理那些没用到的连接。

- 使用任务来并行化不同数据库的连接。使用任务来打开同一个数据库的多个数据库连接可能会获得相当大的性能提升。

A.2 单例模式和服务定位器模式

单例是只有一个实例的类。服务定位器是控制对象分辨率或者在运行时映射接口到特定的执行类型的单例。服务定位器模式支持单元测试与合成或组合应用程序。查询某个提供特别接口的实例的服务定位器，返回的结果是单个共享对象、一个不能共享的实例或者某个池中的缓存实例。这通常被称为“解析”一个实例。

通常，对象的实例是通过类的构造函数创建的。以下的例子说明了这个关系。

```
MyClass variable = new MyClass();
```

相反，如果想要得到这个单一实例，就可以用到以下代码。

```
MyClass variable = MyClass.Instance;
```

在此，**Instance** 是一个类的静态只读属性，而且它每次被调用时，它将会引用 **MyClass** 同一个单例。通过增加这个特殊的 **Instance** 属性，**MyClass** 会被转换成一个单例类。

这个由单例模式返回的实例是延迟求值的。延迟求值意味着实例只有在被需要的时候才会被创建。程序应该避免创建一个既会使用很多资源也会在创建到实际使用时消耗很多时间的对象。

有时候应用单例模式是会受到批评的，因为它会导致状态的共享。如果已经完整阅读完本书，就会知道共享状态是会阻止并行实现的。然而，单例模式是一个可以简单识别全局状态的好方法。要注意，如果在应用程序中看到单例模式和服务定位器模式，那么很有可能会出现共享状态的情

况。因为共享状态很有可能会影响应用程序的扩展性，所以是否可以在简单地应用单例模式或服务定位器模式之前移除对共享状态的依赖是值得考虑的一个问题。

想要了解关于不同方法实现线程安全的单例模式的详情，请参考 A.8 节给出的资源。

A.2.1 使用 LAZY<T>类实现单例模式

.NET Framework 4 使得实现单例模式变得很简单。它提供了一个名为 LAZY<T>的类。以下提供了一个示例。

```
public sealed class LazySingleton
{
    private readonly static Lazy<LazySingleton> instance =
        new Lazy<LazySingleton>(() => new LazySingleton());

    private LazySingleton() { }

    public static LazySingleton Instance
    {
        get { return instance.Value; }
    }
}
```

LAZY<T>类不仅仅处理创建实例值所需要的锁，同时也会从 instance 构造函数中将任何异常抛出给调用代码。这个类使用了 sealed 关键字，以防止类的进一步继承和预期外的修改。

A.2.2 说明

LAZY<T>类提供了一个线程安全的方法来解析类中某个实例，但是它不能保证类的余下部分也是线程安全的。单例模式意味着共享状态，因此可能需要用到进一步的同步机制。

服务定位器模式的实现也需要处理同步问题。如果已经编写了自己的服务定位器或者依赖注入容器，而且想要使得它线程安全，那么既需要保证多个解析不会相互干扰，同时也需要使用锁来保证每次只有一个线程解析一个类型。如果正使用某个第三方依赖注入容器，那么请检查它的解析相关类是不是线程安全的。大部分依赖注入容器，如 Unity Application Block (Unity)，都支持线程安全的解析，但是不支持线程安全的配置。

A.2.3 指南

以下是在并行编程中使用单例模式和服务定位器模式的指南。

- 总的来说，服务定位器模式和单例模式都强烈地表明程序依赖于某个共享资源或者共享数据。可以考虑通过修改算法来消除共享资源或数据在其他地方的调用。
- 共享资源可能会影响并行应用程序的性能，因为想要访问共享资源的任务不得不等待访问机会。这种影响的程度是由任务等待时间的多少来决定的，相对于做任务所花费的时间而言。
- 考虑是否有必要使用单例，单例中的所有数据是否必须齐备。应该只将重要数据放于单例中。
- 评估必须被保护的数据需要多久才能被访问。如果单例模式包含不同种类的数据，且这些数据有着不同程度的访问频率，那么可能需要用到很多锁。
- 如果正使用某个第三方或者自己写的服务定位器模式，那么一定要确保它的对象解析功能是线程安全的。要记住，如果定位器是通过从多个线程中连续访问来保证正确性，那么服务定位器模式可能会影响程序性能。
- 本节中的例子展示了如何创建线程安全的单例模式。设计它们返回的共享对象时，也必须考虑并行访问。

A.3 MVVM

MVP(Model-View-Presenter)和 MVVM(Model-View-ViewModel)都是从用户界面(UI)中分割出的域逻辑。这样的分割可以使应用程序更容易理解和测试。模型封装了应用程序的逻辑和状态。视图模型用绑定模式向视图展示了合适的状态。它通过管理状态和交互逻辑来支持视图。视图模型对底层模型而言充当的是外观模式的角色。它可能会增加视图所需要的额外行为。也可能会实现一个复合模式，方法是使用多于一个模型对象组成自身。

MVVM 对基于 Windows Presentation Foundation (WPF)的应用程序而言是相当有用的，因为它利用了 WPF 对数据绑定的广泛支持这一有利条件。在 WPF 中，视图可以是非常简单的，且它只包含很少代码或不包含代码。视图依赖数据和命令绑定来连接到视图模型。MVVM 模式是表现模型模式的一种特殊形式，它是专门为 WPF 服务的。

MVP 和 MVVM 从 UI 逻辑中分离出域逻辑的关系，但是它们不依赖于数据绑定。在 MVP 中，呈现器充当模型和视图的中介。本节内容只展示 MVVM 模式的例子，没有展示 MVP 模式的，但是大多数的 MVVM 的例子对 MVP 同样适用。欲了解更多关于视图模型和呈现器的信息，请阅读 A.8 节给出的参考资料。

从并行编程者的角度来看，最大的挑战在于，要保证在模型中运行的并行代码可以正确地处理 UI 线程、更新 UI 和用线程安全的方法展示任何数据。本节讨论了如何使用连续的任务来应对这个挑战。在此仍然有其他与并发性和如何建立响应用户需求的多线程 UI 相关的问题，但是这些问题都超出了本书的范围。

A.3.1 示例

第 5 章展示的 Adatum 仪表盘应用程序是并行 MVVM 应用程序的一个例子。要理解这点，请参考在线例子中 Chapter5/A-Dash 项目的代码。图 A.1 说明了这个结构。

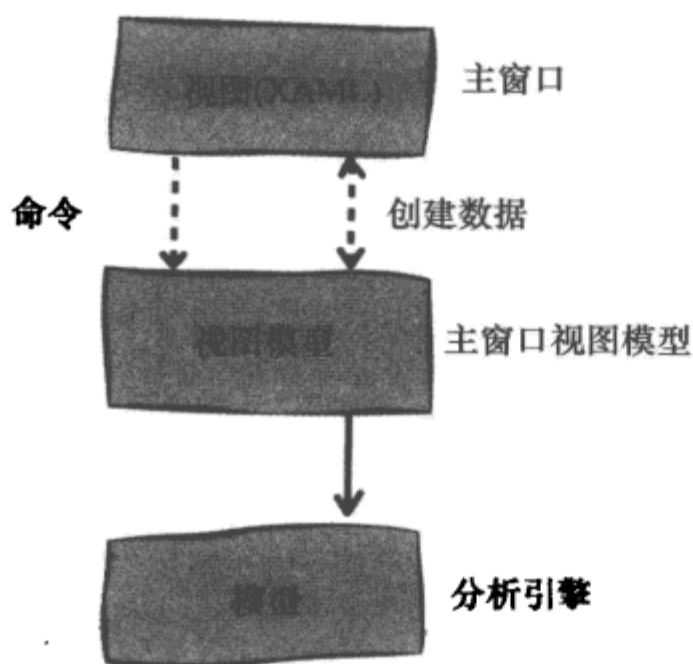


图 A.1 MVVM 模式

职员希望应用程序总是保持能响应的状态，无论计算负荷和 I/O 延迟如何。过长时间的 UI 阻塞是不可接受的。为此，任何一个利用了并行性的优势来实现计算的程序都需要使用后台线程来保证 UI 可以保持响应。Adatum 仪表板的例子说明了如何在 .NET 线程池中执行这个模型以及如何使用视图模型来将模型的状态展示给视图。

DoAnalysisParallel 方法返回了一个 AnalysisTasks 对象，该对象包含了图中任务的每个步骤的 future。例如，CompareModels 属性包含了一个 future，该 future 返回一个市场推荐值。future 是作为 Task<MarketRecommendation>对象来执行的。接下来的代码取回了分析的最后结果。

```
AnalysisTasks tasks = engine.DoAnalysisParallel();
```

这个代码会立即返回一个包含 `Task<>` 对象的 `AnalysisTasks` 对象给每一步的计算。

如果感兴趣，可以检查分析完成前的任何可用的部分和。例如，接下来的代码取回了从网络中载入 NYSE 市场数据的任务所产生的信息。

```
MarketData nyseData = tasks.LoadNyseData.Result;
```

A.3.2 仪表板的用户界面

`future` 和延续任务均被用于依赖于 WPF 的 `Adatum` 仪表板 UI 中。由于 `Adatum` 仪表板 UI 是被设计好的，因此分析中每个步骤的结果都可以被用户当做计算过程来查看。每个步骤都有各自的按钮。当每个结果变得可用时，其对应的按钮会被启用。从 UI 中取消分析是可能的。`Adatum` 仪表板应用程序展示了应用程序如何通过使用延续任务来保持 UI 的更新。

应用程序使用的是延续任务而不是事件处理程序或其他调用机制。事实上，可以把一个延续任务看成一种回调函数。

在此有一个关于通知如何工作的描述。以下的代码用于主窗口视图模型。

```
public class MainWindowViewModel : IMainWindowViewModel
    INotifyPropertyChanged, IDisposable
{
    // ...
    void OnRequestCalculate()
    {
        // ...

        AnalysisTasks tasks = engine.DoAnalysisParallel();
        AddButtonContinuations(tasks);
    }
}
```

```
// ...  
}
```

用户界面使用了 MVVM 模式。主窗口的视图模型有一个 `Calculate Command`，当用户单击 `Calculate` 按钮时，它就会调用 `OnRequestCalculate` 方法。

视图模型要求分析引擎创建一个包含对应于每个分析结果的 `AnalysisTasks` 对象，而不是用 `ThreadPool` 类中 `QueueUserWorkItem` 之类的方法开始后台工作。

这个结构表明了去耦关系。应用程序的分析引擎的开发者不需要知道应用程序的其他部分如何使用分析中的结果。

接下来，处理程序使用 `AddButtonContinuations` 方法创建了特定 UI 的延续任务。接下来的代码展示了这个创建过程。

```
void AddButtonContinuations(AnalysisTasks tasks)  
{  
    AddButtonContinuation(tasks.LoadNyseData,  
        t => { NyseMarketData = t.Result; });  
  
    AddButtonContinuation(tasks.LoadNasdaqData,  
        t => { NasdaqMarketData = t.Result; });  
  
    AddButtonContinuation(tasks.LoadFedHistoricalData,  
        t => { FedHistoricalData = t.Result; });  
  
    AddButtonContinuation(tasks.MergeMarketData,  
        t => { MergedMarketData = t.Result; });  
  
    AddButtonContinuation(tasks.NormalizeHistoricalData,  
        t => { NormalizedHistoricalData = t.Result; });  
  
    AddButtonContinuation(tasks.NormalizeMarketData,
```

```
t => { NormalizedMarketData = t.Result; });

AddButtonContinuation(tasks.AnalyzeHistoricalData,
    t => { AnalyzedHistoricalData = t.Result; });

AddButtonContinuation(tasks.AnalyzeMarketData,
    t => { AnalyzedStockData = t.Result; });

AddButtonContinuation(tasks.ModelHistoricalData,
    t => { ModeledHistoricalData = t.Result; });

AddButtonContinuation(tasks.ModelMarketData,
    t => { ModeledMarketData = t.Result; });

AddButtonContinuation(tasks.CompareModels,
    t =>
    {
        this.Recommendation = t.Result;
        this.StatusTextBoxText = (this.Recommendation == null) ?
            "Canceled" : this.Recommendation.Value;
        this.ModelState = State.Ready;
    });

tasks.ErrorHandler.ContinueWith(
    t =>
    {
        if (t.Status == TaskStatus.Faulted)
            this.StatusTextBoxText = "Error";
        this.ModelState = State.Ready;
    }, TaskScheduler.FromCurrentSynchronizationContext());
}
```

AddButtonContinuations 方法会在每个任务完成时使用一个特殊的调度程序来创建自动在 UI 线程上运行的延续任务,且为视图准备好了结果。

`TaskScheduler` 类的 `FromCurrentSynchronizationContext` 方法的结果是一个 `TaskScheduler` 对象，该对象允许它的任务只在当前线程(本例中即 UI 线程)所规定的同步环境中运行。

每个延续任务都会在基于任务的结果的视图模型上设置一个属性。当最后的 `CompareModels` 任务完成时，UI 视图模型会随着最终推荐值而更新。在视图模型上注册的回调函数会通知 UI 有变化，因此这些变化将会反映在 UI 上。

最后一个延续任务是错误处理程序。如果某个应用程序发生错误了，那么 `Adatum` 仪表板应用程序会使用一个专为错误处理程序工作的延续任务来观察该异常。进行错误处理的延续任务随着在 UI 线程上运行的第二个延续任务而更新 UI。

辅助方法 `AddButtonContinuation` 是 `ContinueWith` 方法的一个简单应用，它可以创建延续任务。以下代码展示了这个方法。

```
void AddButtonContinuation<T>(Task<T> task,
                              Action<Task<T>> action)
{
    task.ContinueWith(
        action,
        CancellationToken.None,
        TaskContinuationOptions.OnlyOnRanToCompletion,
        TaskScheduler.FromCurrentSynchronizationContext());
}
```

在将并行性融入应用程序之前，`Adatum` 使用后台工作线程处理仪表板中计算密集的部分。然而，对 `Adatum` 仪表板应用程序仍然有一些要求(例如 UI 中 WPF 的使用)，以便延续任务更恰当地工作。

`future` 模式对 `Adatum` 仪表板有效的一个原因是，它满足了 WPF 强加的线程关联要求。一些框架对它们操作的对象设置了这个约束。例如，在

WPF 中，必须让你曾经创建的 UI 对象的所有方法在同一个线程上运行。应用程序一般都使用 Dispatcher 类来安排将要在 UI 线程上处理的工作的程序。欲了解更多信息，请阅读 A.8 节推荐的参考资料。

future 模式使得处理线程关联变得更容易。延续任务可以由在某个特殊的线程上运行该延续任务的任务调度程序所配置。先行任务不需要和延续任务在同一个线程上运行。这个特性允许 Adatum 的开发者将计算密集型的工作分散在多个内核中，并保证出现在 UI 中的计算值不会违背 WPF 线程关联的约束。

任务使得要满足线程的关联约束更容易。

A.3.3 指南

以下是一些使用 future 模式和延续任务的指南。

- 使用与模型相连的延续任务来更新视图模型。这个方法可以帮助你从视图中解耦模型。
- 使用不可变类型将结果从延续任务传递给 UI 线程。这样更容易保证代码是正确的。
- 如果想要在后台执行多个不同任务，且想要在结果被传回给 UI 线程之前合并结果，应该使用延续任务而不是后台工作线程。
- 使用正确的任务调度程序以确保对线程关联的 UI 对象的修改仅发生在主要的 UI 线程上。

A.4 不可变类型

当编写并行代码时，要减少一个任务与另一个任务之间相互干扰的可能，不应该在任务中分享任何有可变字段的对象，而应该使用不可变类型（有时候也被称为纯函数性数据结构）。可以使用不可变类型实现（不可变）值对象模式。

不要忽视不可变类型对并行编程的重要性。它们可以提高程序的可靠性和性能。

一个熟悉的关于不可变数据类型的例子是.NET 的 `string` 类。`System.String` 类有很多关于创建字符串的操作,但是没有一个允许修改字符串。实际上,那些所谓修改字符串的操作都会创建新的字符串。另外一个不可变类型的例子是.NET 的 `System.Uri` 类。该类的实例相当于一个统一资源识别器(URI),该类的方法和属性便于你创建 URI 并查询其组元。完全不用担心使用 URI 时会有另一个线程修改它。

不可变类型是一种非常有用的可扩展的分享类型。例如,当你追加一个字符串时,返回结果是一个包含了该附加串的字符串。原始串是没有被修改的。这就意味着总是可以在并行代码中使用一个特殊串值,而不用担心另一个任务或者线程可以修改它。不可变类型和.NET Framework 并发集合类的结合产物是相当有用的。想要详细了解为什么不可变类型对于并行编程是一个很好的方法,请阅读 1.2.3 节。

一个类型必须满足以下条件才能被认为是不可变类型。

- 该类型的所有字段无法在构造函数之外被修改。这就意味着一个对象的状态在它被构造之后就不能被改变了。C#的 `readonly` 关键字可以在编译时强制此项。
- 该类型必须只包含不可变类型字段。
- 该类型必须只能从其他不可变类型中继承。
- 该类型不能被可变类型继承。尽管这个条件看起来是不必要的,但是它可以防止可变类型继承不可变类型,并防止可变类型通过重载一个虚拟方法来改变不可变类型的行为。可以用 `sealed` 关键字标记不可变类型以达到上述目的。
- 该类型的实例直到构造完成之后才能发布自身的引用。如果一个对象在构造期间暴露了自身的引用(它的 `this` 引用),那么很可能该引用会在构造完成之前被另一个线程所访问。紧接着第二个线

程可能会观察到这个在构造期间内部状态正改变的对象。换句话说，不变的假设可能会不成立。

如果只要求一个对象是观察性不变的，那么以上这些条件是可以适当放宽的：这种对象表面上看上去好像是不可变的，可是实际上它包含了一些私有可变的状态。如果选择用这个方法，那么实现必须保证对象的私有可变状态的变化是以线程安全的方式完成的。`ReadOnlyCollection<T>`和`Lazy<T>`类都是观察性不可变类型的例子。

C#中的对象不是默认为不可变的，但可以实现不可变类型。F#之类的编程语言都包含创建不可变类型的内置支持。

A.4.1 示例

在第5章中 `Adatum` 仪表板例子用过的很多类型都是不可变类型。这些类型的值传递了财务分析的结果。例如，这有个 `StockData` 类的实现。该类的实例表示了一个财务资产的价格历史。`Adatum` 仪表板例子在任务中传递 `StockData` 和 `StockDataCollection` 的对象。在此不需要任何锁来实现，因为该类本身就是不可变的，因此不会受到其他副作用的影响。

```
public sealed class StockData
{
    readonly string name;
    readonly ReadOnlyCollection<double> priceHistory;

    public string Name
    {
        get { return name; }
    }

    public ReadOnlyCollection<double> PriceHistory
    {
        get { return priceHistory; }
    }
}
```

```
}

public StockData(string name, double[] priceHistory)
{
    this.name = name;
    this.priceHistory = new
        ReadOnlyCollection<double>(priceHistory);
}
// ...
}
```

`StockData` 类使用了 `readonly` 关键字来保证该类不会随意改变它的字段。如果尝试改变一个 `readonly` 字段，编译器就会发出一个错误提示。

尽管 `StockData` 类不会改变字段，但还是要将 `Name` 属性定义为一个带私有设置函数的自动属性。以下是一个例子。

```
class StockData2
{
    public string Name { get; private set; }
    // ...
}
```

自动属性的一个优点是它们很简洁而且很容易设置属性。它的一个缺点就是必须手动确保这个类的其他操作不会改变该属性。设置 `setter` 为私有的，可以防止该类的外部调用者改变这个属性。

A.4.2 不可变类型作为值类型

值类型使用复制语义的方式来工作。如果两个这样的值类型所对应的所有的域都相等，那么它们的值就会是相等的。如果两个值相等，那么它们的哈希码也相等。`.NET` 类型中 `System.Double` 是值类型的一个例子。可以比较引用类型和值类型。`C#` 中的类都是引用类型。在默认情况下，只有当某个类的实例都是 `new` 操作符的同一个调用的结果时，类的实例

才是相等的。

尽管不可变类型的实现使用的是引用类型,但是你通常想要不可变类型能表现得跟值类型一样。例如,可能需要使用类型的实例作为某个字典的键值。`System.String` 类是引用类型的一个例子,它在很多方面都效仿值类型的行为。(可以通过调用 `Object.ReferenceEquals` 方法来忽略 `String` 类的“值类型”的抽象层。)

可以通过执行一个使用了结构相等(一个域一个域地比较)而不是引用相等(`Object.ReferenceEquals` 方法使用的测试)的 `Equals` 方法使得不可变类型可以表现得跟值类型一样。如果在类型中执行 `Equals` 方法,必须还要执行 `GetHashCode` 方法,以便于相等的值总是会有相同的哈希码。

使用等于(==)操作符是值得推荐的方法。以下是一个例子。

```
public sealed class StockData
{
    // ...

    public static bool operator ==(StockData a, StockData b)
    {
        if (System.Object.ReferenceEquals(a, b))
            return true;
        if (((object)a == null) || ((object)b == null))
            return false;
        if (a.Name != b.Name)
            return false;
        if (a.PriceHistory.Count != b.PriceHistory.Count)
            return false;
        for (int i = 0; i < a.PriceHistory.Count; i++)
        {
            if (a.PriceHistory[i] != b.PriceHistory[i])
                return false;
        }
    }
}
```

```
    }  
    return true;  
}  
  
public static bool operator !=(StockData a, StockData b)  
{  
    return !(a == b);  
}  
  
public override bool Equals(object obj)  
{  
    if (obj == null)  
        return false;  
    return Equals(obj as StockData);  
}  
  
public bool Equals(StockData d)  
{  
    if (d == null)  
        return false;  
    return (this == d);  
}  
  
public override int GetHashCode()  
{  
    int result = name.GetHashCode() ^ priceHistory.Count;  
    for (int i = 0; i < priceHistory.Count; i++)  
        result ^= priceHistory[i].GetHashCode();  
    return result;  
}  
}
```

欲了解更多关于等于操作符的内容，请阅读 A.8 节推荐的参考资料。

A.4.3 复合值

不可变类型并不局限于带有固定数值的域的记录。不可变类型也可以实现复杂结构，如表、树和不可变或者纯函数性数据类型的序列。例如，在不可变类型的树上进行“插入节点”的操作会返回一棵新的树。

首先，你可能会想到如果表和树表现得跟不可变类型一样可能是不会很有效的。然而，在此有一些实现技术并不需要你复制整个结构的每个插入或删除操作。实现中使用相当多的结构共享是有可能的。想要了解更多信息，请阅读 A.8 节推荐的参考资料。

A.4.4 指南

以下是关于使用不可变类型的指南。

- 如要在线程中共享数据可考虑使用不可变类型。可以将它们与本附录下一节中要讨论的共享的数据结构相结合。
- 对于更复杂的类型，可以考虑实现“可冻结的”或“冰棒式”不可变性。这样的类型在创建时是可变的，但是当使用者调用一个“冻结”方法时它们就变得不可变了。WPF 应用了这个方法，并提供一个 `System.Windows.Freezable` 基础类。你也可以使用一个生成器模式来提供一个可以在冻结前递增地创建不可变值的方法。

欲了解更多关于 C#语言版如何创建不可变类型的内容，请阅读 A.8 节推荐的参考资料。

A.5 共享的数据类

本书讨论过的很多模式都在刻意减少共享数据的数量。.NET Framework 4 提供了一个 `System.Collections.Concurrent` 命名空间，该空间

包含了很多线程安全的数据结构。如果认为应用程序需要共享状态，那么在自己编写实现方法之前，请回顾一下 .NET Framework 4 提供的集合。编写既正确又性能好的共享数据结构并不容易。表 A.1 描述了 .NET Framework 4 中的此类数据结构。

表 A.1 .NET Framework 4 中的共享数据结构

| 类 型 | 描 述 |
|--------------------------------------------|-----------------------------------------------------------------------------------|
| <code>BlockingCollection<T></code> | 这个数据结构实现了有界的和无界的生产者/消费者系统。在生产者和消费者的设计中使用它来连接任务。第 7 章中的流水线模式使用阻塞集合的方法连接各流水线阶段。 |
| <code>ConcurrentBag<T></code> | 这个数据结构是个对象的无序集合，它对存储那些与顺序无关的数据是很有用的。第 6 章中的代码样例包含了关于存储无序结果于并发包中的例子。 |
| <code>ConcurrentDictionary<T></code> | 这个数据结构是一个存储和取回键/值组合的并发字典。 |
| <code>ConcurrentQueue<T></code> | 这个数据结构是一个并发的且可扩展的先进先出(FIFO)无阻塞的队列。可以使用它来连接不同的任务或者给任务分发工作。第 6 章讨论的变化形式使用了队列给任务提供值。 |
| <code>ConcurrentStack<T></code> | 这个数据结构是一个并发的且可扩展的后进先出(LIFO)的栈。如果添加的上一个元素对接下来的操作很重要，则使用此数据结构。 |

`System.Collections.Generic` 命名空间中的集合不是线程安全的。如果集合被多个线程共享，必须使用 `System.Collections.Concurrent` 中的集合。

指南

以下是几点关于使用共享数据的指南。

- 在使用共享数据之前请再三考虑。本书中描述的很多模式都想要限制共享，但是又避免完全限制。
- 尽可能使用共享数据集合而不要使用锁。
- 尽可能使用 .NET Framework 4 提供的共享数据类而不要编写自定义类。

A.6 迭代器

迭代器使得程序可以控制另一个方法的流。迭代器跟我们在计算机科学中所说的协同程序是类似的。C#语言内部支持迭代器的编写。以下是一个例子。

```
public static IEnumerable<string> GetImageFileNames(  
    string sourceDir, int maxImages)  
{  
    var names = GetImageFileNamesList(sourceDir, maxImages);  
    while (true)  
    {  
        foreach (var name in names)  
            yield return name;  
    }  
}
```

该例会产生一个无限长的文件名列表。

示例

当合并并行循环时，迭代器是很有用的。事实上，一个自定义迭代器就是并行循环的一种扩展方法。例如，下列代码展现了一个二叉树。

```
class Tree<T>  
{  
    public Tree<T> Left, Right;  
    public T Data;  
}
```

可以在 `Tree<T>` 类中实现一个自定义迭代器。该迭代器可以用于某个并行循环来访问树的节点。下列代码展示了此过程。

```
public IEnumerable<Tree<T>> Iterate<T>()
{
    var queue = new Queue<Tree<T>>();
    queue.Enqueue(this);
    while (queue.Count > 0)
    {
        var node = queue.Dequeue();
        yield return node;
        if (node.Left != null) queue.Enqueue(node.Left);
        if (node.Right != null) queue.Enqueue(node.Right);
    }
}
```

这个自定义迭代器可以跟 `Parallel.ForEach` 方法一起使用。

```
Tree<T> myTree = ...

Parallel.ForEach(myTree.Iterator(), node =>
{
    // 并行中的进程节点
});
```

另一种更先进的变形可能会根据子树创建分割区。例如，可以实现一个 `Partitioner` 对象，该对象将树划分成很多子树，每个子树的根对应于原来的树中某个确定深度的节点。如果，子树内存中的存储位置提高了内存的缓存性能，那么这个方法会变得相当有效。

A.7 列表和枚举

.NET 类型 `IList<T>` 定义了索引列表的功能。`IEnumerable<T>` 类型被用于未被索引化的迭代。

可以在 `IList` 和 `IEnumerable` 都可用且控制并行循环中使用哪个。只有在某些特定的不常见的情况中才需要考虑这些。

应该注意这两个类型是如何与并行循环相互作用的。在某些罕见的情况下，并行循环中默认的处理 `IList<T>` 的方法可能不是你想要的。这种情况会发生在 `IList<T>` 的实现有不利的随机访问性能特征时或者由于多线程同时尝试执行延迟加载而产生竞争条件时。

可以重写对提供 `IList<T>` 接口的源的 `Parallel.ForEach` 默认处理方法。

`Parallel.ForEach` 方法需要它的源(即传入的参数)提供 `IEnumerable<T>` 接口；并且，它也会检查源是否提供了 `IList<T>` 接口。如果提供，那么 `Parallel.ForEach` 默认可以使用这个接口。在大多数情况下，在更有效的分区策略中应使用 `IList<T>` 访问结果集合中的元素，因为它会提供对集合中元素随机的(也就是索引化的)的访问。相比之下，`IEnumerable<T>` 只支持使用能获取连续元素的 `MoveNext` 方法来遍历集合的访问。在几乎所有情况下，默认的行为会获得更好的性能。

有一些提供 `IList<T>` 的类型在某种程度上使得并行的索引化变成一个昂贵甚至错误的操作。对于这些类型，`MoveNext` 是一个更好的访问方法。因为可以使用一个 `Partitioner` 对象迫使 `Parallel.ForEach` 在 `IList<T>` 可用的情况下仍然使用 `IEnumerable<T>` 接口。下面是一个例子。

```
IEnumerable<T> source = ...;

// 将会经常使用源的 IEnumerable<T> 实现
Parallel.ForEach(Partitioner.Create(source),
    item => { /*... do work ... */ });
```

`System.Data.Linq.EntitySet<TEntity>` 是一个应该在并行迭代中使用 `IEnumerable<T>` 的类型的例子。其原因是这个类型的延迟加载语义。(如果两个线程尝试并发访问索引器，那么 `EntitySet<TEntity>` 实例可能会被损坏。)

A.8 扩展阅读

以下是关于本附录所讨论的主题的一些扩展内容的参考来源。

A.8.1 结构模式

- ADO.NET 最佳实践实例, 请参见 MSDN 上的“Best Practices for Using ADO.NET” : <http://msdn.microsoft.com/en-us/library/ms971481.aspx>。
- ADO.NET 的使用指南, 请参见 MSDN 上“Improving .NET Application Performance and Scalability”中的第 12 章“Improving ADO.NET Performance” : <http://msdn.microsoft.com/en-us/library/ff647768.aspx>。
- ADO.NET 的实体框架概述, 请参见 MSDN 上的“ADO.NET Entity Framework” : <http://msdn.microsoft.com/en-us/library/bb399572.aspx>。
- 关于库, 请参见 Addison-Wesley Professional 2002 年出版的 *Patterns of Enterprise Application Architecture*, 作者是 Martin Fowler。
- 关于 Windows Azure, 包括并行数据库访问的库的讨论, 请参见 Windows Azure Guidance 网站上的 patterns & practices: <http://wag.codeplex.com/>。

A.8.2 单例模式

- “Gang of Four”模式的原文, 请参见 *Design Patterns: Elements of Reusable Object-Oriented Software*(Gamma, Erich, Richard Helm, Ralph Johnson, John M. Vlissides. *Design Patterns:*

Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, 1994)。

- 使用 C#实现单例模式和其他设计模式的详情, 请参见 *C# 3.0 Design Patterns* (Bishop J. *C# 3.0 Design Patterns*. O'Reilly, 2008)。
- 想要了解更多细节, 请参见 Duffy 著作(2008)的第 10 章, 其中包含对单例模式的详细描述(Duffy J. *Concurrent Programming on Windows*. Addison-Wesley, 2008)。
- 关于统一依赖注入容器的记载和容器的讨论, 请参见 MSDN 上的“Unity Appliaction Block” : <http://msdn.microsoft.com/unity>。
- 关于延迟初始化的详情, 请参见 MSDN 上的“Lazy Initialization” : <http://msdn.microsoft.com/en-us/library/dd997286.aspx>。

A.8.3 MVVM

- 关于 MVVM 的详细讨论, 请参见 MSDN 杂志上的文章“WPF Apps With The Model-View-ViewModel Design Pattern” : <http://msdn.microsoft.com/en-us/magazine/dd419663.aspx>。
- 关于 MVP 模式, 请参见 MSDN 杂志上的文章“Model View Presenter” : <http://msdn.microsoft.com/en-us/magazine/cc188690.aspx>。
- 关于 WPF 线程模型和如何用 Dispatcher 类来更新 WPF 对象的概述, 请参见 MSDN 杂志上的文章“Build More Responsive Apps With The Dispatcher” : <http://msdn.microsoft.com/en-us/magazine/cc163328.aspx>。
- 关于 BackgroundWorker 类的详情(包括如何使用它和相关例子), 请参见 MSDN 上的“BackgroundWorker Class” : <http://msdn.microsoft.com/en-us/library/system.componentmodel.backgroundworker.aspx>。

A.8.4 不可变类型

- Eric Lippert 的博客包括了一系列出色的关于用 C#语言实现不可变性的专栏: <http://blogs.msdn.com/b/ericlippert/archive/tags/immutability/>。
- Joe Duffy 的博客也从并行角度阐述了对不可变性的需求: <http://www.bluebytesoftware.com/blog/2007/11/11/ImmutableTypesForC.aspx>。
- 更多关于相等性的细节, 请参见 MSDN 上的“Guidelines for Overriding Equals() and Operator == (C# Programming Guide)”:
[http://msdn.microsoft.com/en-us/library/ms173147\(VS.90\).aspx](http://msdn.microsoft.com/en-us/library/ms173147(VS.90).aspx)。
- 关于实现不可变类型, 包括复杂结构(如序列、表和树)的详情, 请参见 *Purely Functional Data Structures* (Okasaki C. *Purely Functional Data Structures*. Cambridge University Press, 1998)。
- 关于用 C#实现不可变类型的序列、集合和映射, 请参见 CodePlex 上的 NModel framework: <http://nmodel.codeplex.com>。



附录 B 调试和分析并行应用程序

Visual Studio 2010 开发系统调试器包含了两个有助于并行编程的窗口：“并行堆栈”窗口和“并行任务”窗口。另外，Visual Studio 2010 的 Premium 和 Ultimate 版本还包含另一个分析工具。本节附录给出了一些例子，这些例子展示了如何使用这些窗口和分析器来可视化一个并行程序，以及如何确定程序是否按预期运行。积累了这方面的一些经验之后，就能利用这些工具确认和解决问题。

B.1 “并行任务”窗口和“并行堆栈”窗口

在 Visual Studio 环境中，打开并行引导示例解决方案。设定第 7 章中的项目 ImagePipeline 为启动项目。打开 ImagePipeline.cs，找到方法

单上的“继续”(Continue)以加载一些图片], 然后流水线开始填充图片, 并且其他的任务也开始运行。这一过程如图 B.1 所示。回想一下, 每个任务都单独运行在一个线程上。“并行任务”窗口显示了线程的任务分配情况。当任务内联产生时, 一个线程可以运行多个任务, 所以当它的一个任务被阻塞时, 仍有可能处于运行状态。

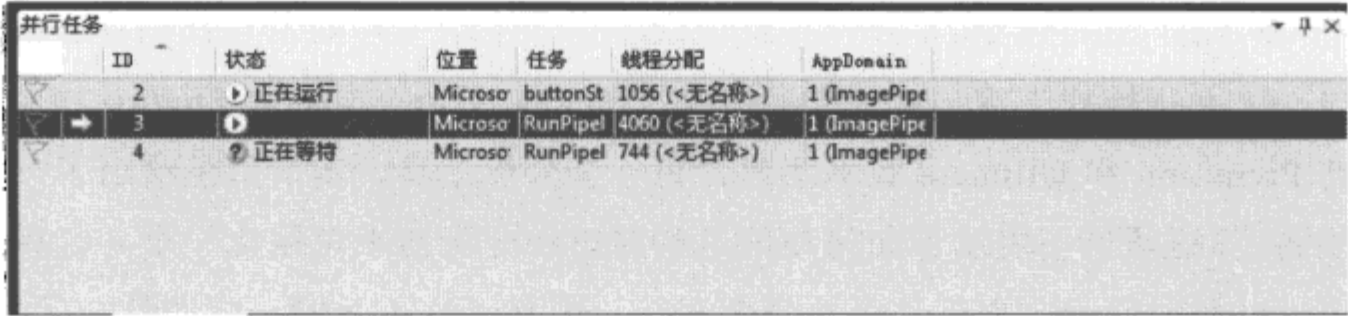


图 B.1 “并行任务”窗口

将鼠标指向“调试”菜单上的“窗口”, 选择“并行堆栈”(Parallel Stacks)。在“并行堆栈”窗口左上角的下拉菜单中, 选择“任务”(Tasks), 然后在“并行任务”窗口上右击, 选择“显示外部代码”(Show External Code)[可能需要先禁用“仅我的代码”(Enable Just My Code)。可以通过以下方式找到该选项, 选择“工具”(Tools)|“选项”(Options)|“调试”(Debugging)。该窗口显示了每一个任务的堆栈]。图 B.2 中, 该窗口的内容被放大了(使用窗口左侧的缩放控制, 或通过鼠标滚轮来操作), 所以只能看到两个栈的内容, 事实上所有的栈都能访问到。

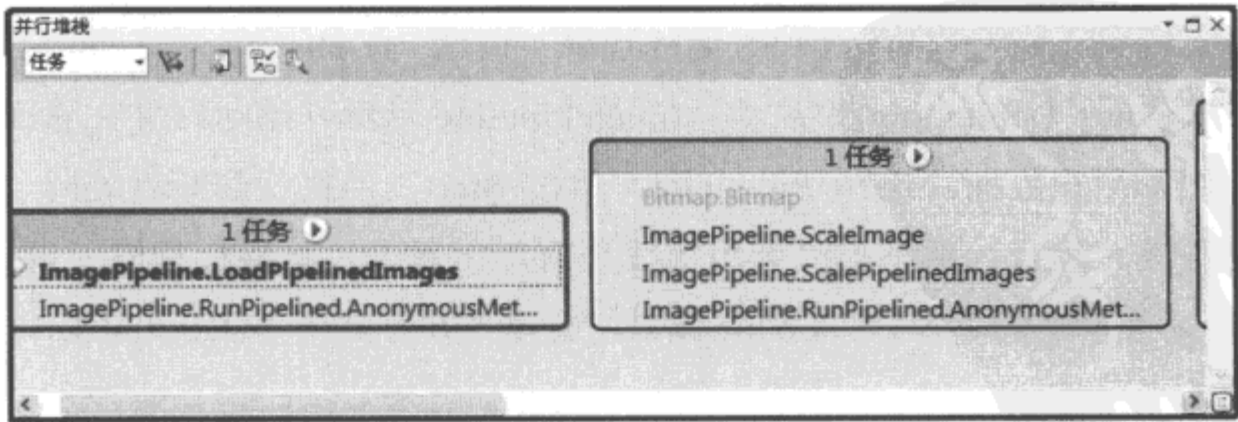


图 B.2 “并行堆栈”窗口

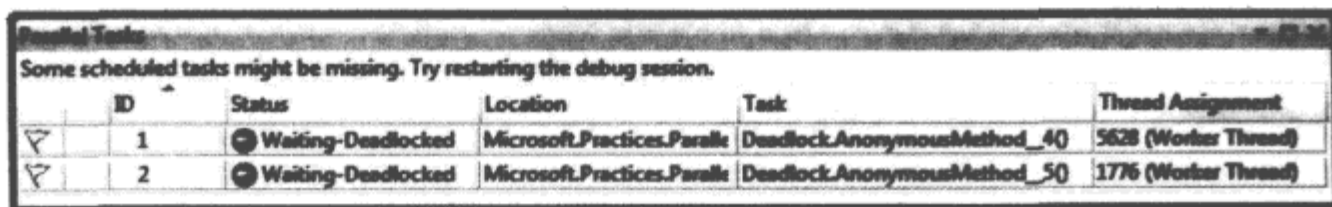
继续按 F5 键，每个窗口的内容都会随着流水线缓冲区在空和填充这两个状态之间的变化而变化。这是预期的行为，因此不会出现错误提示。这些窗口也可能揭示一些异常的行为，这些行为能帮助你确认和解决性能问题和同步错误。例如，“并行任务”窗口和“并行堆栈”窗口有助于确认常见的并发问题，如死锁。示例如下。

```
static void Deadlock()
{
    object obj1 = new object();
    object obj2 = new object();

    Parallel.Invoke(
        () => {
            for (int i = 0; ; i++) {
                lock (obj1) {
                    Console.WriteLine("Got 1 at {0}", i);
                    lock (obj2) Console.WriteLine("Got 2 at {0}", i);
                }
            }
        },
        () => {
            for (int i = 0; ; i++) {
                lock (obj2) {
                    Console.WriteLine("Got 2 at {0}", i);
                    lock (obj1) Console.WriteLine("Got 1 at {0}", i);
                }
            }
        }
    );
}
```

以上代码是一个经典的死锁例子。两个任务都在试图以一种会导致循环的顺序获得锁。这样将最终导致死锁。在运行这些代码的过程中的某个时候，应用程序会停止继续往下执行(这一现象会很明显，因为控制台不

再有新的输出)。这时候, 如果使用“调试”菜单上的“全部终止”(Break All)选项, 并且打开“并行任务”窗口, 将会看到图 B.3 所示的结果。



| ID | Status | Location | Task | Thread Assignment |
|----|--------------------|--------------------------------|----------------------------|----------------------|
| 1 | Waiting-Deadlocked | Microsoft.Practices.Paralle... | DeadlockAnonymousMethod_40 | 5628 (Worker Thread) |
| 2 | Waiting-Deadlocked | Microsoft.Practices.Paralle... | DeadlockAnonymousMethod_50 | 1776 (Worker Thread) |

图 B.3 “并行任务”窗口显示死锁

B.2 Concurrency Visualizer

Visual Studio 2010 分析器中包含了 Concurrency Visualizer。它显示了并行代码运行时如何使用资源: 使用了多少内核, 在内核之间分布的线程数以及每个线程的活动。这些信息可以帮助确定并行代码是否按照你所设计的那样运行, 同时它还能帮助你诊断性能问题。

Concurrency Visualizer 的工作过程有两个阶段: 数据收集阶段和可视化阶段。在数据收集阶段, 启动数据收集并运行应用程序。在可视化阶段, 检查收集到的数据。本附录在一台双核计算机上使用 Concurrency Visualizer 来分析第 7 章的 ImagePipeline 例子。

首先完成数据收集阶段。为此, 必须以管理员身份运行 Visual Studio, 因为数据收集使用的是内核级记录。在 Visual Studio 中打开并行引导示例解决方案。有多种方式来启动数据收集。其中一种方式是选择 Visual Studio “调试”菜单上的“启动性能分析”(Start Performance Analysis), 性能向导开始运行。然后选择“并发”(Concurrency), 选择“可视化多线程应用程序的行为”(Visualize the behavior of a multithreaded application)。该向导的下一个页面将显示解决方案, 该解决方案当前在 Visual Studio 中已打开。选中想分析的项目, 即 ImagePipeline。单击“下一步”(Next)。向导的最后一个页面会询问在向导结束之后是否立即开始分析。默认情况下, 此复选框处于选中状态。单击“下一步”, 出现 Visual Studio 分析器

窗口，这意味着分析正在进行。ImagePipeline 例子开始运行并且打开了它的图形用户界面窗口。为了最大限度地提高处理器的使用率，选中“负载均衡”(Load Balanced)选项，然后单击“启动”。为了收集到大量数据来进行可视化，必须让图像计数器(在图形界面上)至少达到 20。然后单击 Visual Studio 分析窗口上的“停止分析”(Stop Profiling)。

在数据收集阶段，性能分析器会频繁地抽取样本数据(被称为快照)，这些样本数据记录了当前运行的并行代码的状态。每一次数据收集都会写一些数据文件，包括一个.vsp 文件。一次单一的数据收集过程可能会写入达百兆的文件。由于某些不可控因素，如其他进程在同一台计算机上运行，同一个程序的不同运行过程中所收集的数据是不同的。

可以在文件可用的任何时间运行可视化阶段，且不需要以管理员身份来运行 Visual Studio 就可以做到。有多种方法来启动可视化阶段。可以请求性能向导当数据收集结束时立即开始可视化。或者，仅仅需要打开任何 Visual Studio 中的.vsp 文件。如果选中第一个选项，当数据收集和分析完成后，将会看到一个摘要报告。摘要报告显示了你可以看到的不同的视图。这些视图包括一个线程图，一个 CPU 利用率图和一个内核图。

图 B.4 为线程图。每一个任务都在一个线程中执行。Concurrency Visualizer 显示出每个任务的线程(记住，由于内联任务的关系，可能一个线程会对应多个任务)。

Concurrency Visualizer 屏幕包含的许多细节信息在该图中没有清楚地显示出来，因为缩小了尺寸且没有显示全部颜色。本附录的全彩屏幕截图版本可以从 CodePlex 站点得到，其网址为 <http://parallelpatterns.codeplex.com/>。

图 B.5 为 CPU 使用率视图。CPU 使用率视图显示了整个应用程序使用的处理器(逻辑内核)数，且其结果会随着时间的变化而变化。运行此例子的计算机上有两个逻辑内核。其他与该应用程序无关的进程也显示出来。对于每个进程而言，有一个图标显示每个时间点其使用的处理器数量。

为了使处理器容易区分，每个进程的图形区域用不同的颜色显示(某些颜色可能无法准确地体现在此图中)。某些数据点显示一个分数，不是 0、1 或者 2，因为每个点代表采样区间的平均值。

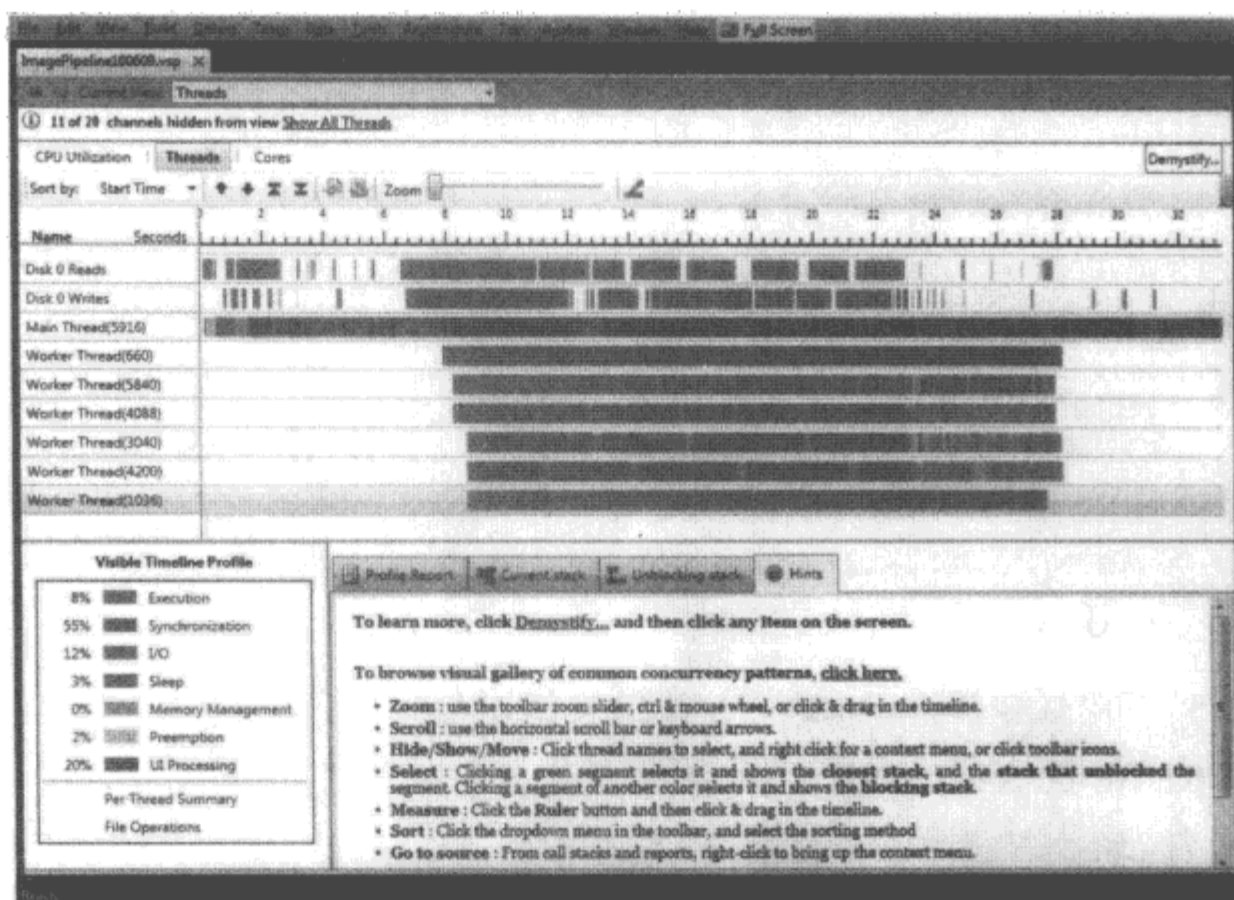


图 B.4 Concurrency Visualizer 的线程图

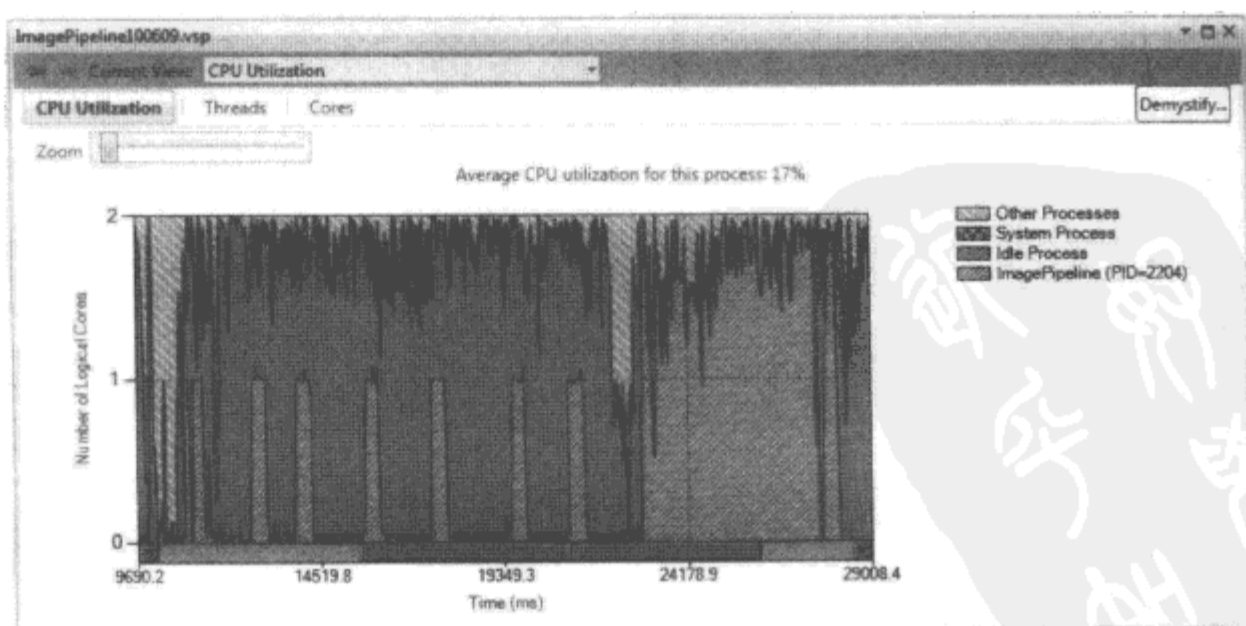


图 B.5 CPU 使用率详细视图

在这个特定的数据收集过程中,图形用户界面显示出,第一次缓慢出现了几张图片,剩下的图片出现得较为迅速(这一现象并不会在每次运行过程中出现)。视图反映了这种现象。运行过程早期(从时间尺度上来说,大约在 20 000 毫秒以前)应用程序运行出现“井喷”(指极快)现象,当它加载图片时会填充流水线。当流水线中有一些图片之后(即大约 20 000 毫秒以后),流水线任务能并行运行并且应用程序使用两个逻辑内核。这一视图还显示了“井喷”现象之间的区间(20 000 毫秒之前),此时应用程序还没有获得处理器。在这些间隔期间应用程序被阻塞或者被其他进程抢占。

图 B.6 为内核视图。该视图显示了应用程序如何使用可用的内核。每个内核都有一张时间表,一个指示每个线程运行时刻的颜色编码带(不同的颜色表示每一个线程)。在 10 到 22 这一时间区间,应用程序运行出现“井喷”,空区间便是当时没有与该进程有关的作业运行在内核之中。在 22 到 28 时刻之间,流水线被填充了,更多的任务比内核中的进程优先级高。每个内核都出现线程切换并且图下方的表格显示有大量的上下文切换。

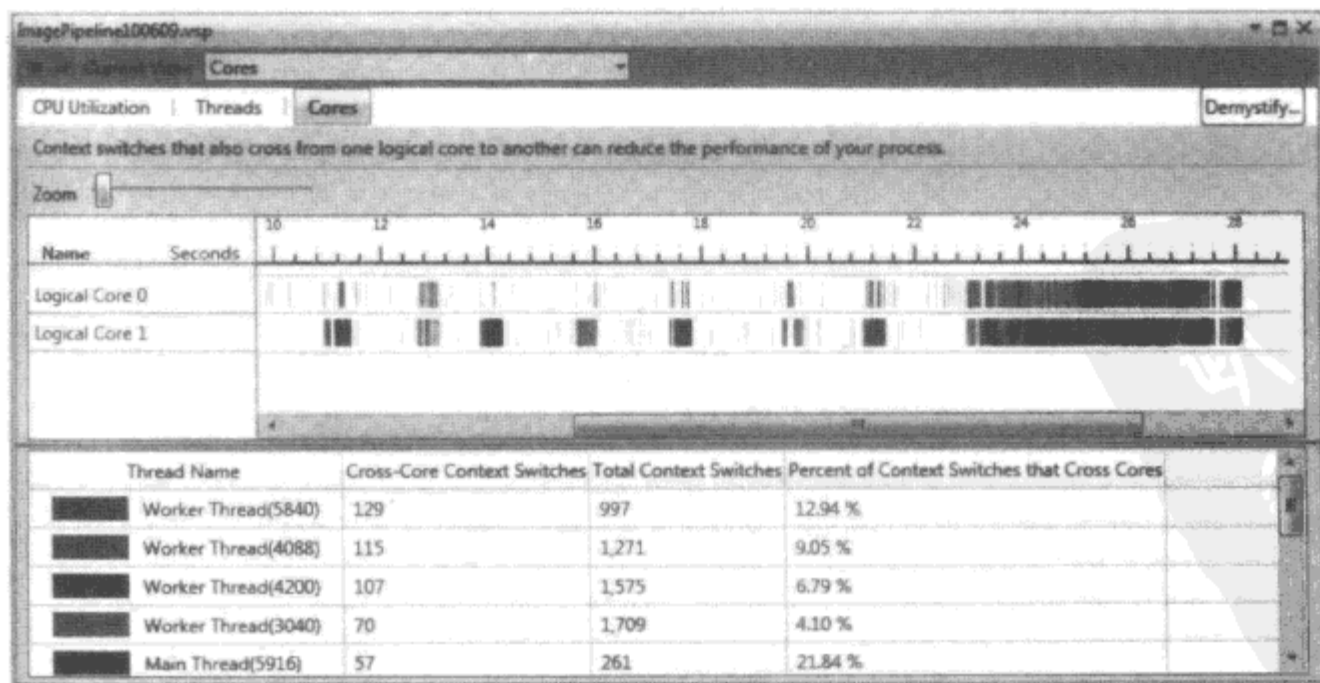


图 B.6 内核详细视图

图 B.7 为线程图。线程图显示每个线程如何分配时间。有一个通过色带来表示不同类型活动的时间线。例如，红色表示线程正在同步(等待状态)。该视图初始时显示线程池中的空闲线程。[可以通过如下操作隐藏它们：右击视图，然后选择“隐藏”(Hide)]。从视图中可以看出主线程贯穿整个活动过程；绿色表示用户界面活动。

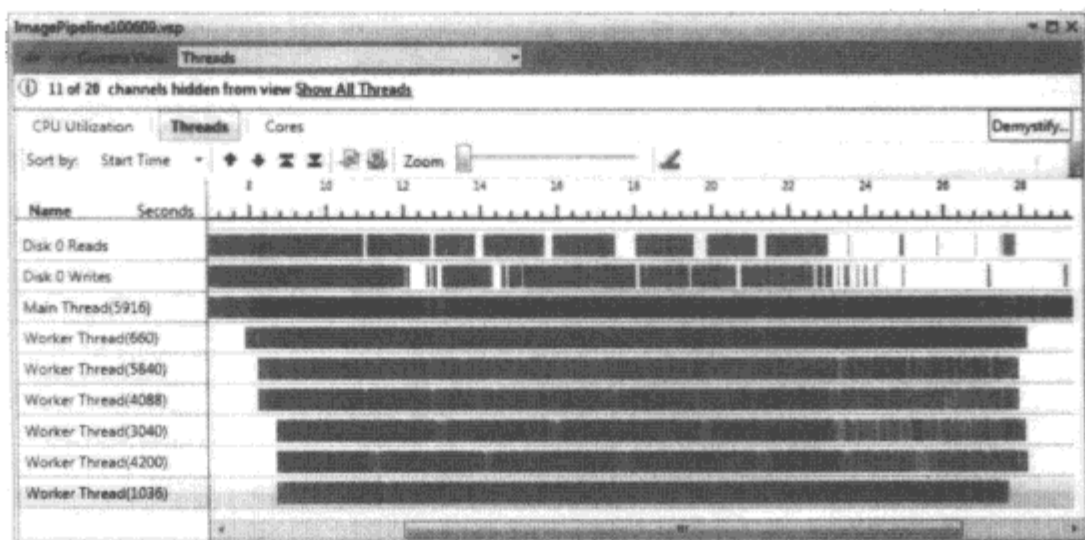


图 B.7 线程详细视图

流水线线程在时刻 22 之前处于“突发”状态，此时它们在等待流水线被填充，故在运行和同步之间交替。过了 22，一些流水线线程执行频繁而其他的一些几乎连续执行。流水线线程数大于内核数，所以一些流水线线程不得不在执行和被抢占之间切换。

可以使用 Scenario 库来标识复杂应用程序的不同阶段。下列代码就是一个例子。(Scenario 库在 MSDN Code Gallery 网站可以免费下载。更多相关信息，请参见 MSDN 的“Scenario Marker Support”：<http://msdn.microsoft.com/en-us/library/dd984115.aspx>。)

```
Scenario.Scenario myScenario = new Scenario.Scenario();
myScenario.Begin(0, "Main Calculation");

// Main calculation phase...
myScenario.End(0, "Main Calculation");
```

这些标记会在线程图和 CPU 使用率图中显示出来。不要使用太多标记，因为它们很容易削弱视觉效果且使它失去易读性。工具可能会隐藏一些标记以提高可读性。可以使用缩放功能来增加放大倍率，这样可以看到特定部分的隐藏标记。

B.3 视觉模式

本书讨论的模式主要着眼于潜在并行性的表达方式。然而，在并行开发中，还有一些有用的模式。人类的思维擅长识别视觉模式，**Concurrency Visualizer** 就利用了这一点。可以学着识别一些常见的视觉模式，这些视觉模式通常会在应用程序出现特殊的性能问题时出现。本节描述的视觉模式将会帮助你识别和解决超额申请、锁争用和负载失衡等问题。

B.3.1 超额申请

当线程数量超过了逻辑处理器的数量时，超额申请问题就会出现。超额申请会因为频繁的上下文切换导致性能下降，而且每一次上下文切换都会消耗处理时间并且会弱化内存缓冲带来的优势。

Concurrency Visualizer 便于我们发现超额申请问题，因为超额申请问题会导致分析器追踪区域出现大量黄色。黄色意味着一个线程被抢占了（即线程被切出内核）。当追踪时，下面的代码会产生一个超额申请问题的典型写照。

```
static void Oversubscription()
{
    for (int i = 0; i < (Environment.ProcessorCount * 4); i++)
    {
        new Thread(() => {
            // Do work
            for (int j = 0; j < 1000000000; j++) ;
        })
    }
}
```

```
        }).Start();  
    }  
}
```

图 B.8 显示了该函数在一个四核系统中运行一次的线程图。它产生了一个非常独特的模式。

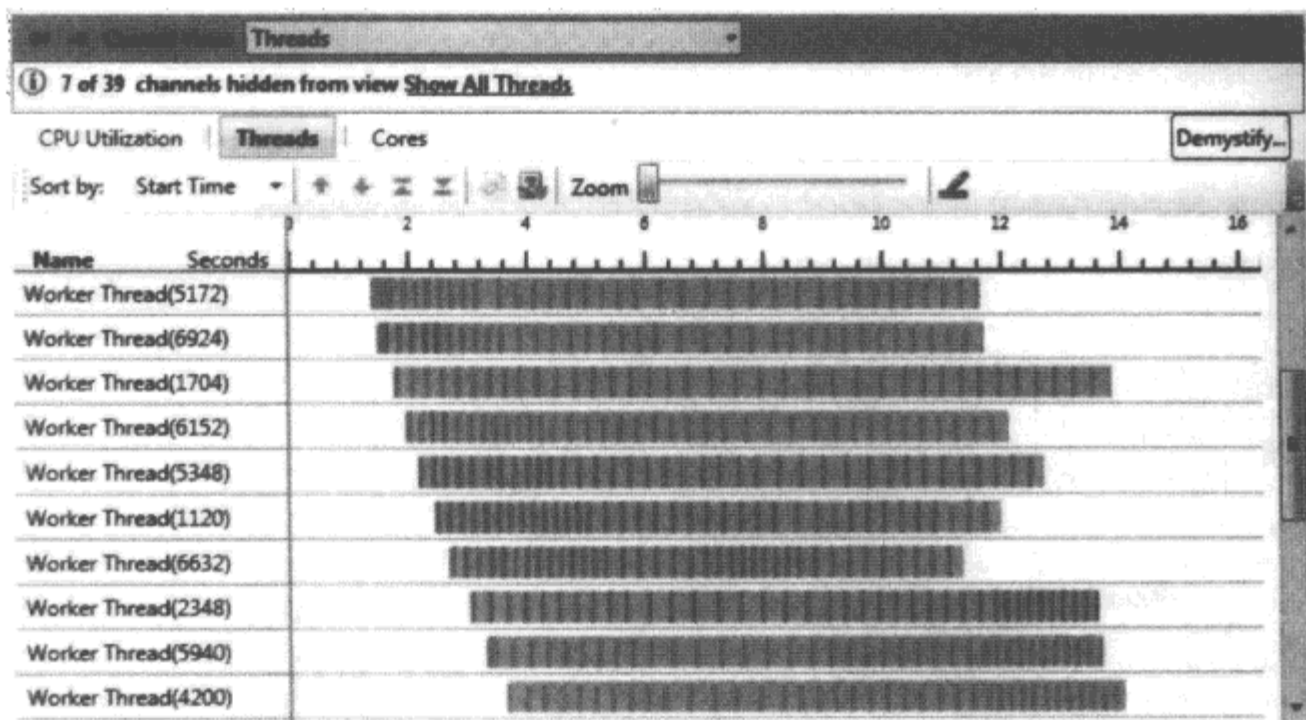


图 B.8 显示超额申请的线程图

B.3.2 锁争用与顺序化

当一个线程视图试图获得已经被另外一个线程持有的锁时,就会发生争用。在许多情况下,这一结果会一直持续到先持有锁的线程释放锁。Concurrency Visualizer 的线程图用红色表示阻塞。它往往是性能下降的迹象。在极端情况下,一个应用程序可能被一个或多个锁完全顺序化,即使正在使用多线程。

接下来的方法会产生一个锁保护,它导致,即使是多线程环境下,程序也会出现严重的锁争用和顺序化现象。锁保护是一个性能问题,当多个线程争用一个频繁共享的资源时,该问题就会发生。


```
static void LockContention()
{
    object syncObj = new object();
    for (int p = 0; p < Environment.ProcessorCount; p++)
    {
        new Thread(() => {
            for(int i=0; i<50; i++)
            {
                // Do work
                for (int j = 0; j < 1000; j++);

                // Do protected work
                lock (syncObj)
                    for (int j = 0; j < 100000000; j++);
            }
        }).Start();
    }
}
```

图 B.9 显示了这些代码在 Concurrency Visualizer 中产生的线程图。

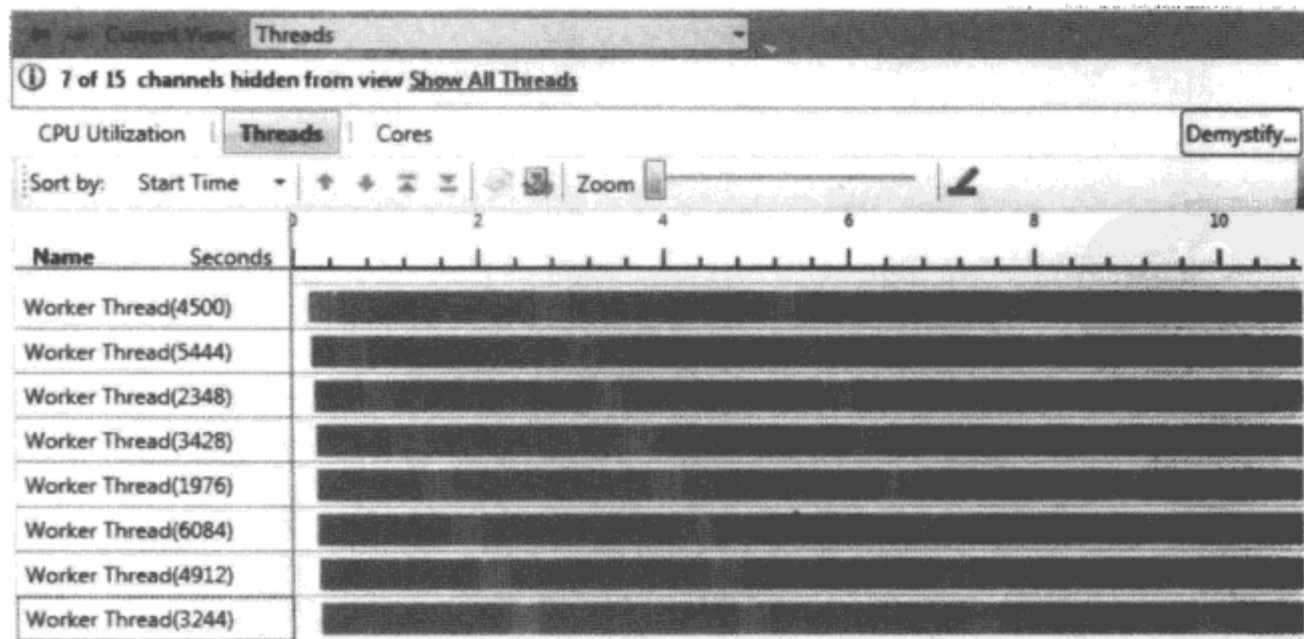


图 B.9 线程图显示锁保护

B.3.3 负载失衡

当作业不均地分布在所有的并发操作涉及的工作线程中时,负载失衡就发生了。负载失衡意味着系统利用不足,因为当某些线程执行的时候,一些线程却闲置着。由负载失衡产生的视觉模式在许多 **Concurrency Visualizer** 中都是可以辨别的。下面这段代码用来产生负载失衡。

```
static void LoadImbalance()
{
    const int loadFactor = 10;

    ParallelEnumerable.Range(0, 100000).ForAll(i =>
    {
        for (int j = 0; j < (i * loadFactor); j++) ;
    });
}
```

尽管 **.NET Framework 4** 支持的大部分并行机制都是使用动态分区在工作任务之间分配作业的,但 **PLINQ** 的方法 **ParallelEnumerable.Range** 却使用静态分区。这个例子运行在有 8 个逻辑内核的系统上,会导致元素 **[0,12499]** 被一个任务执行,而元素 **[12500,24999]** 被另一个任务执行,以此类推。工作体仅仅在 0 和当前索引值之间迭代,这意味着需要做的工作量与索引成正比。处理低范围的线程将会比处理高范围的线程少很多任务。图 B.10 为 **Concurrency Visualizer** 的 CPU 使用率视图,描述了这种现象。

当该方法刚开始执行时,系统中所有的 8 个逻辑内核都被使用了。然而一段时间之后,由于每个内核都完成了各自的工作,其使用率开始下降。这产生了一个“阶梯模式”,当线程完成各自部分的作业后便被释放。图 B.11 所示的线程视图证实了这点。

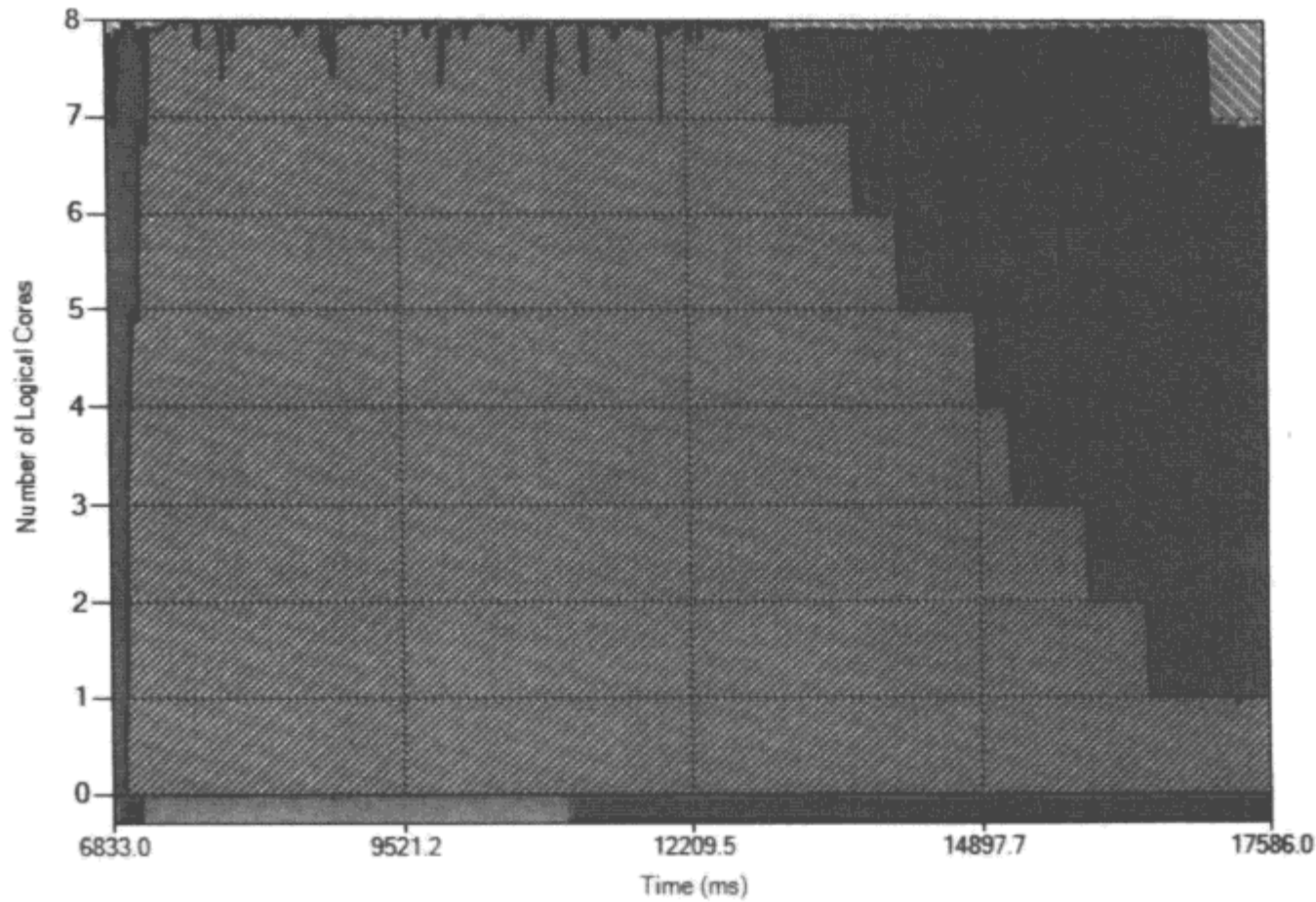


图 B.10 负载失衡下的 CPU 视图

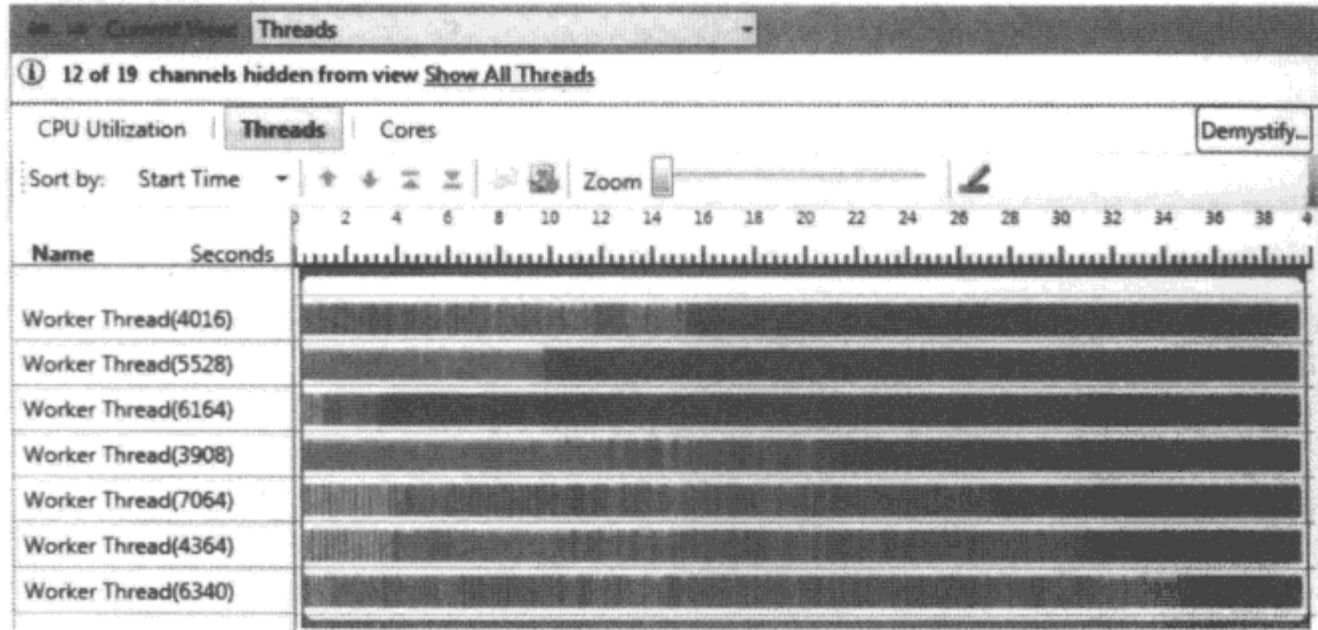


图 B.11 负载失衡的线程图

从线程图可以看出，完成部分工作后，所有的工作线程都进入空闲状态，等待 CLR 工作线程 6340 完成余下的工作。

B.4 扩展阅读

MSDN 上的并行性能分析博客讨论了许多技术和例子。MSDN 还提供了关于 Scenario 库的信息。

- MSDN 上关于 Visual Studio 2010 Parallel Performance Analysis 的博客：<http://blogs.msdn.com/b/visualizeparallel/>。
- MSDN 上的“Performance Tuning with the Concurrency Visualizer in Visual Studio 2010”：<http://msdn.microsoft.com/en-us/magazine/ee336027.aspx>。
- MSDN 上的 Scenario 主页：<http://code.msdn.microsoft.com/scenario>。



附录 C 技术概览

本附录描述了一些微软提供的并行计算资源，它们没有体现在本书前面的内容中。后面的“扩展阅读”中包含了一些 URL，可以通过访问它们获取更多信息。图 C.1 展示了不同的供给源与它们之间的关系。(请注意 UMS 线程是一个用户模式的调度线程。应用程序可利用该机制调度其自己的线程。)

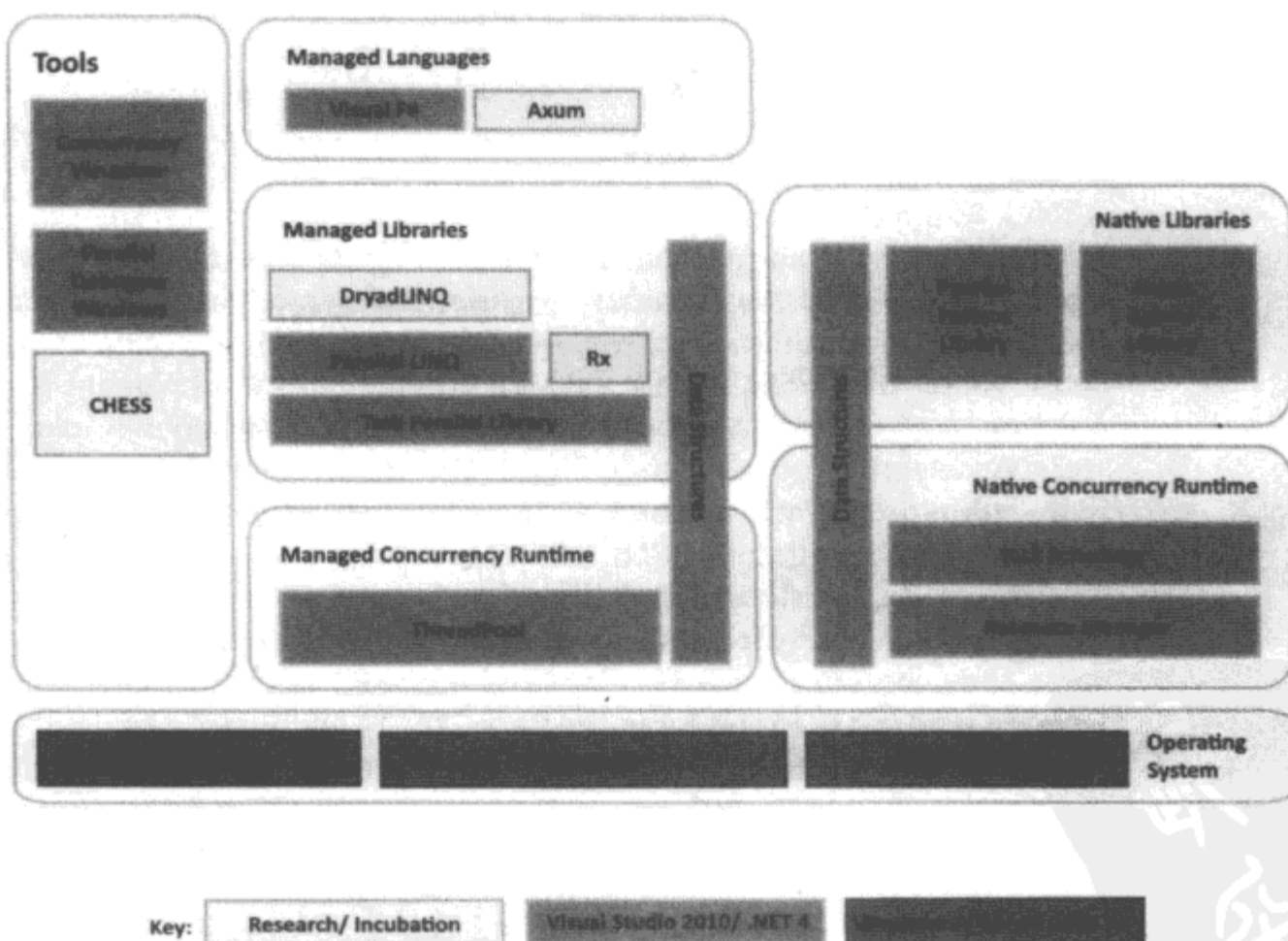


图 C.1 微软并行编程资源

本书前面描述了任务并行库(TPL)和并行 LINQ(PLINQ)，它们均为 Microsoft .NET Framework 4 附带的管理库。它们依赖于.NET ThreadPool

类和 .NET Framework 4 的并发数据结构。

F#语言也是 Visual Studio 2010 开发系统附带的。它展示了一个比 TPL 或 PLINQ 功能更多的方法，它强调数据类型的不可变性。然而，F#运行时库建立在 TPL 之上并与其集成，F# PowerPack 包含了建立在 PLINQ 之上的并行支持。本书中的示例已被移植到 F#中。

对于 Visual C++ 开发系统的开发者来说，该系统包含有并行模式和异步代理库。它们使用本地并发运行时，包括一个强大的调度器和资源管理器，这些都用于在多核架构中高效执行本地并行工作负载。本书的 Visual C++版本很快就会面世了。

Visual Studio 2010 包含了一些调试和分析并行应用程序的工具。有关如何使用它们的示例，请参见附录 B。还可以使用 Microsoft Research 的 CHES 工具来侦测并行代码中的漏洞。

除了 Visual Studio 2010 和 .NET Framework 4 附带的技术，还有一些其他有趣的研究项目专注于并行和并发，其中包括 Axum 语言、.NET 的 Reactive Extensions(Rx)和 DryadLINQ。Axum 是一种并发编程语言，为了增加程序的安全性、响应性、可扩展性和开发效率，它以解耦、行为和消息传递为原则。Rx 是一个库，它使用多个集合来组成异步和基于事件的程序。DryadLINQ 是一个编程环境，可用于编写运行在大型 PC 机群上的大规模数据的并程序。Accelerator(没有在图 C.1 中展示)是一个使用数据并行编程模型的库，既可用于图形处理单元 GPU，又可用于多核 CPU。

Axum、Rx、Accelerator 和 DryadLINQ 都是处于研究中的项目，Microsoft 并没有承诺附带它们。但是，它们包括的许多新思想和方法定能引起读者的兴趣。我们倡议读者下载和评估它们，并向各个团队反馈意见。

除了 Axum 和 DryadLINQ，所有以上技术主要用于单个计算机上的并行。Windows High Performance Computing (HPC) Server 2008 则以服务

器机群为目标，支持在多台计算机上横向扩展。尽管集群计算的技术各不相同，但本书讨论的一些基本模式仍然是适用的，例如并行任务和并行合并。

扩展阅读

- MSDN 并行计算开发中心包含了托管和本地并行运行时的并行开发工具，还包括了支持编写并行程序的 Visual Studio 2010 工具：<http://msdn.microsoft.com/concurrency>。
- 欲了解更多 F# 的信息，包括其语言参考和演练，请访问 Microsoft F# 的开发中心：<http://msdn.microsoft.com/fsharp>。
- Windows HPC Server 的站点上包含了 Windows HPC Server 2008 的产品信息和开发资源：<http://www.microsoft.com/hpc>。
- 有关 Microsoft Research 项目 Accelerator 和 DryadLINQ 的详细信息，请通过以下网址查看：<http://research.microsoft.com/accelerator> 和 <http://research.microsoft.com/en-us/projects/DryadLINQ>。
- 有关 Microsoft DevLabs 项目的详情，包括 Rx、Axum 和 CHESS 的描述和下载，请访问 <http://msdn.microsoft.com/devlabs>。



术 语 表

汇总异常 包含一个或多个内部错误的异常，而不是仅仅只有一个内部异常。

合并计算 将多个数据项汇总到一个结果。

Alpha 混合 将不同的图像以半透明层的方式叠加到一张图像上。

先行任务 该任务的完成能够启动一个延续任务。先行任务通常是一个 future，其结果值作为延续任务的输入。

异步操作 启动时不阻塞当前线程的控制权的操作。

后台线程 当进程关闭时所停止的线程。后台线程的运行不能保证进程的运行。线程池中的线程都是后台线程，与前台线程对应。

关卡(barrier) 一个同步点，所有参与的线程都必须停止并等待，直至所有的进程都到达为止。.NET 中的 `System.Threading.Barrier` 类支持关卡。

阻塞 当等待事件或条件时，中止执行任务。

阻塞集合 集合中的移除操作将被阻塞，直到有数据可移除；其他操作将被阻塞，直到有空间可用。.NET 中的 `System.Collections.Concurrent.BlockingCollection<T>` 类支持阻塞集合。

捕获变量 用于 lambda 表达式的变量，在 lambda 表达式之外定义。lambda 表达式可更新捕获变量。

集群 由多台计算机组成的并行计算系统，机器之间通过网络连接，不是指单个物理处理器上的多个内核。

闭包 捕获变量的 lambda 表达式。

并发 同时处理多个活动。并发使得程序可以即时对外界做出响应，其目标是减少延迟。并发可以通过异步操作或线程实现，需要线程能够轮流地在处理器上执行。与并行对比。

Concurrency Visualizer Visual Studio 分析器的附加组件，展示并行程序的执行和性能信息。

上下文切换 一个线程在处理器上停止执行时，另一个线程恢复执行。当处理器上的线程过多时容易引起过多的上下文切换，这将导致性能降低。

延续任务 当先行任务完成时自动启动的任务。

控制流 任务按照某算法的步骤执行时的协调基础，如在并行循环中。

内核 物理处理器上执行指令的部分。大多数最新的物理处理器模型的内核都超过一个，因此能并行执行任务。

协同程序 一个广义子程序(方法)，支持不同的入口且返回不止一次，每次产生不同的返回值。在 C# 中，迭代器通常作为使用 yield 关键字的协同程序实现。

数据流 当数据可用时执行任务的协调基础，如在流水线或任务图中。

数据并行 并行处理的一种形式，相同的计算以不同的数据并行执行。Microsoft .NET Framework 中，Parallel.For 和 Parallel.ForEach 方法以及 PLINQ 技术支持数据并行。与任务并行对比。

数据分割 将数据集合分割成块，以便使用数据并行。

数据竞争 超过一个并发线程不同步地读写数据。

死锁 由于系统在等待一个无法发生的条件，因而执行停止后无法恢复。当某个线程持有另一个线程所需的资源时，可能导致死锁。

分解 将一个问题分解为更小的块。在并行处理中，可根据数据或任务进行分解。

并行度 可同时执行的最大并行任务数。并行度可以通过 `WithDegreeOfParallelism` 方法在 `PLINQ` 中设置，也可以通过 `MaxDegreeOfParallelism` 选项在并行循环中限定。

依赖 某个操作使用了另一个操作的结果。当操作之间存在依赖时，它们不能并行执行。与独立对应。

依赖注入(DI)容器 通过传入(注入)依赖构建对象图的框架或库。对象生命周期由容器处理，而不是使用它的对象。

双重检查锁定 首先测试一个条件，只有当条件为真时，获取一个锁并再次测试该条件，以确定是否更新共享数据。该策略通常能够避免当锁不可用时仍然去获取锁所需要的高开销操作。

动态分割 数据分割的各个部分作为并行任务执行。与静态分割对应。

前台线程 保持进程运行的线程。前台线程停止时，进程关闭。与后台线程对应。

分支/合并(fork/join) 使用任务并行的一种并行计算模式。分支发生在任务启动时，合并发生在所有任务结束时。

future 返回一个值的任务。在 .NET Framework 中，`future` 由 `Task<TResult>` 类实现。

粒度 任务中的分区或工作中的数据数量。也可以说是数据分区或任务的数量。粗粒度表示分区或任务很少，细粒度表示有很多小的分区或任务。

硬件线程 内核中的执行流水线。同步多线程(有时称为超线程)允许同时在一个内核上执行多个硬件线程。可以将每个硬件线程看作一个独立的逻辑处理器。

不可变 数据创建之后不能修改。例如，.NET 字符串就是不可变的。与可变相对应。

不可变类型 这种类型的实例是不可变的。其实例是纯函数性数据结构。

独立的 操作时不需要使用其他操作的结果。独立操作可以并行执行。与依赖相对应。

内部异常 包含在另一个异常中，并在另一个异常中抛出的异常。

lambda 表达式 一个包含表达式或语句的匿名函数。

延迟初始化 数据直到需要时才初始化。在.NET 中，延迟初始化由 `Lazy<T>` 和 `Lazy Initializer` 类支持。

活锁 执行没有中断但没有朝目标前进。

负载均衡 不同的任务分配到的工作量差不多，以便有效利用处理器资源。与负载失衡对应。

负载失衡 不同的任务分配到的工作量不同，以至于有些任务无事可做，因而没有有效利用处理器资源。与负载均衡对应。

锁 一种同步机制，确保某个时刻只有一个线程可以执行某段特定代码。

锁保护 多个任务重复竞争同一个锁。多次获取锁失败可能导致性能低下。

逻辑处理器 与某个硬件线程相关的处理器。在.NET 中，`System.Environment.ProcessorCount` 返回逻辑处理器的数量。与物理处理器对应。

多核(manycore) 多核(multicore)，通常有超过 8 个逻辑处理器。

映射 并行计算中，多个任务在不同的数据上独立地进行相同的转

换。数据并行的一个例子。

映射/简化 一种并行编程模式，紧接着数据并行阶段(映射)之后有一个合并阶段(简化)。

内存关卡 一条机器指令，强制内存操作之间的顺序关系。确保在关卡之前的内存操作先发生，在关卡之后的内存操作后发生。

多核 超过一个内核，可执行并行任务。大多数最新的物理处理器模型都是多核的。

重数 某元素出现在多重集中的次数。

多重集 可能包含重复元素的无序集合。集合中的每个元素都与一个重数(或次数)关联，重数表明它出现的次数。与集合对比。

多重集合并 将多重集的元素合并，将各元素的重数相加。

可变的 数据创建后可以修改。例如，.NET 数组是可变的。与不可变对应。

嵌套并行 并行编程结构出现在另一个并行编程结构中。在.NET 中，在 Parallel.For 循环中使用 Parallel.For 循环时，它们得互相协调，以便共享线程资源。

节点 集群中的一台计算机。

非阻塞算法 允许多个任务共同解决同一个问题，且任务之间不会阻塞的算法。

对象图 由互相引用的对象组成的数据结构。对象图通常是共享的可变数据，有可能使并行编程复杂化。

重叠 I/O 当其他任务正在执行时继续执行(或等待)的 I/O 操作。

超额申请 线程数超过了可用的处理器数。超额申请可能导致性能低下，因为需要花费过多上下文切换的时间。

并行 在多个线程上编程，这些线程可以同时多个处理器上执行。其目标是增加吞吐量。与并发对比。

分割 将数据划分为几部分，以便使用数据并行。

物理处理器 处理器芯片，也被称为一个包或套接字。大多数最新的物理处理器模型的内核都超过一个，每个内核的逻辑处理器都超过一个。

流水线 由一连串生产者/消费者组成，每个消费者使用前一个生产者的输出。

优先级反转 一个较低优先级的线程在运行，而较高优先级的线程在等待。这种情况发生在低优先级的线程持有高优先级线程的资源时。

进程 运行中的应用程序。进程可以并行运行，进程之间互相是隔离的(通常不共享数据)。一个进程可以包含几个(或多个)线程或任务。在.NET 中，进程可以被 `System.Diagnostics.Process` 类监测和控制。与线程和任务对比。

分析器 收集和展示性能分析信息的工具。`Concurrency Visualizer` 就是并发和并行程的分析器。

纯函数 没有副作用(没有更新数据或产生输出)的函数(或方法、操作)，仅返回一个值。

纯函数性数据结构 只能由纯函数访问的数据结构。不可变类型的实例。

竞争 计算结果依赖于语句执行的顺序，而执行的顺序没有被控制或同步。

竞争条件 竞争出现的条件。竞争条件通常是异常，优秀的编程实践可以防止它们出现。

递归分解 在并行编程中，一个任务可启动更多任务。

简化 数据通过相关的操作进行合并的一种合并方式，并行中通常可以进行大量简化。

循环法 一个调度算法，每个线程在重复的循环中以固定的顺序运行，因此在每个循环中每个线程仅运行一次。

可伸缩的 一种并行计算模式，性能随着处理器的增加而提高。

信号灯 一种同步机制，确保不超过指定个数的线程同时执行一段特定的代码。与锁对比。

串行化 顺序执行，而非并行。

集合 没有重复元素的非排序集合。与多重集对比。

共享数据 多于一个线程使用的数据。访问可变共享数据需要进行同步化。

单线程数据类型 非线程安全的类型。除非用户代码中有额外的同步化机制，否则不能由多个线程访问。

同步多线程处理(SMT) 在同一个内核上执行多个线程的技术。

套接字 物理处理器。

推测执行 即使有可能不需要任务的结果，仍然执行该任务。

静态分割 数据分割时，在程序执行前已经选择好了块。与动态分割对应。

同步化 协调线程的动作，确保正确的结果。锁即同步机制的一个示例。

任务 工作的一个并行单元。任务在线程上执行，但任务并不等同于线程，它是更高层次的抽象。在.NET中，推荐用 `System.Threading.Tasks` 命名空间中的任务并行库针对任务进行并行编程。与线程、进程对比。

任务图 当任务提供的结果作为其他任务的输入时，可看作一个有向图。节点表示任务，边表示任务的输入输出值。

任务内联 由于某个正在执行的任务请求另一个任务同步运行，导致一个线程上有多个任务同时执行。

任务并行 并行处理的一种形式，不同的计算在不同的数据上并行执行。在.NET 中，`Parallel.Invoke` 和 `Task.Factory.StartNew` 方法支持任务并行。与数据并行对比。

线程 语句的一个执行序列。一些(或多个)线程可以在一个进程中运行。线程之间不是孤立的，一个进程中的所有线程之间共享数据。线程上可运行一个任务，但它不等同于任务，它是更低层次上的抽象。在.NET 中，线程由 `System.Threading` 命名空间支持。与进程、任务对比。

线程关联 多个操作必须在某个特定线程上执行。例如，Windows Presentation Foundation(WPF)控制只能从创建它的线程上访问它。

线程注入 向线程池中增加线程。

线程池 由.NET 管理的线程集合，避免创建和销毁线程的开销。任务通常由线程池中的线程运行。

线程饥饿 由于线程池中无可用的线程，所以任务无法运行。

局部线程状态 只能由一个线程访问的变量。锁或其他同步机制不需要安全访问局部线程状态。在.NET 中，局部线程状态由 `ThreadStatic` 属性和 `ThreadLocal` 类支持。

线程安全类型 可由多个线程并发使用的类型，不需要在用户代码中加入额外的同步化机制。线程安全类型确保其数据一次只能由一个线程访问，其操作都是遵循多线程的原子性操作。在.NET `System.Collections.Concurrent` 命名空间提供了一些线程安全集合类型。与单线程数据类型对比。

断读 读取某个变量需要多条机器指令，而另一个任务在读取指令之间写入该变量。

断写 写入某个变量需要多条机器指令，而另一个任务在写入指令之间读取该变量。

元组 包含一个有序的元素列表的未命名记录。在.NET 中，元组由 Tuple 类支持。

两步舞(two-step dance) 当唤醒线程需要获得某个锁时，标记某个持有该锁的事件。唤醒线程仍需再次等待。这可能引起上下文切换和性能低下。

申请不足 任务的数量比可用的处理器少，因此处理器有闲置的。

虚拟内核 逻辑处理器。

volatile 关键字 该关键字告诉 C#编译器字段可以被多线程、操作系统或其他硬件修改。

work stealing 策略 为了保持忙碌，执行排列在其他线程后面的线程。

XAML .NET Framework 编程模型中使用的声明性标记语言，用于具体说明.NET 应用程序的用户界面(UI)。



参 考 文 献

- Albahari J, Albahari B. *C# 4 in a Nutshell*. fourth edition. O'Reilly, 2010.
- Bishop J. *C# 3.0 Design Patterns*. O'Reilly, 2008.
- Boodhoo J. Model View Presenter. <http://msdn.microsoft.com/en-us/magazine/cc188690.aspx>.
- Campbell C, Veanes M, Huo J, Petrenko A. Multiplexing of Partially Ordered Events. In: TestCom 2005. Springer Verlag. 2005-06. <http://research.microsoft.com/apps/pubs/default.aspx?id=77808>.
- Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters. In: *OSDI'04: Sixth Symposium on Operating System Design and Implementation*. 2004. 137-150.
- Dufy J. *Concurrent Programming on Windows*. Addison-Wesley, 2008.
- Dufy J. Immutable types for C#. <http://www.bluebytesoftware.com/blog/2007/11/11/ImmutableTypesForC.aspx>.
- Fowler M. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003.
- Gamma E, Helm R, Johnson R, Vlissides J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

- Hoag J E. A Tour of Various TPL Options. 2009-04. <http://blogs.msdn.com/b/pfxteam/archive/2010/04/19/9997552.aspx>.
- Jacky J, Veanes M, Campbell C, Schulte W. *Model-Based Software Testing and Analysis with C#*. Cambridge University Press, 2008.
- Leijen D, Schulte W, Burckhardt S. The design of a task parallel library. In: Arora S, Leavens G T, ed. *OOPSLA 2009: Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 2009. 227-242.
- Lippert E. Fabulous Adventures in Coding. <http://blogs.msdn.com/b/ericlippert/archive/tags/immutability/>.
- McCool M. Structured Patterns for Parallel Computation. 2009-12. <http://www.ddj.com/go-parallel/article/showArticle.jhtml?articleID=223101515>.
- Mattson T G, Sanders B A, Massingill B L. *Patterns for Parallel Programming*. Addison-Wesley, 2004.
- Okasaki C. *Purely Functional Data Structures*. Cambridge University Press, 1998.
- Smith J. WPF Apps With the Model-View-ViewModel Design Pattern. <http://msdn.microsoft.com/en-us/magazine/dd419663.aspx>.
- Toub S. Patterns of Parallel Programming: Understanding and Applying Parallel Patterns with the .NET Framework 4 and C#. 2009. <http://www.microsoft.com/downloads/details.aspx?FamilyID=86b3d32b-ad26-4bb8-a3ae-c1637026c3ee&displaylang=en>.
- Toub S. Custom Parallel Partitioning With .NET 4. 2010-04-26.

<http://www.drdobbs.com/visualstudio/224600406>.

- Wildermuth S. WPF Threads: Build More Responsive Apps With The Dispatcher. <http://msdn.microsoft.com/en-us/magazine/cc163328.aspx>.

其他在线资源

- 加速器: <http://research.microsoft.com/en-us/projects/accelerator/>.
- The ADO.NET Entity Framework Overview. [http://msdn.microsoft.com/en-us/library/aa697427\(VS.80\).aspx](http://msdn.microsoft.com/en-us/library/aa697427(VS.80).aspx).
- Best Practices for using ADO.NET. <http://msdn.microsoft.com/en-us/library/ms971481.aspx>.
- DevLabs 门户网站: <http://msdn.microsoft.com/en-us/devlabs/default.aspx>.
- DryadLINQ: <http://research.microsoft.com/en-us/projects/DryadLINQ/>.
- Parallel Programming with Microsoft .NET. <http://parallelpatterns.codeplex.com/>.
- Guidelines for Overriding Equals() and Operator == (C# Programming Guide). [http://msdn.microsoft.com/en-us/library/ms173147\(VS.90\).aspx](http://msdn.microsoft.com/en-us/library/ms173147(VS.90).aspx).
- How to: Create a Task Scheduler That Limits the Degree of Concurrency. <http://msdn.microsoft.com/en-us/library/ee789351.aspx>.
- Improving ADO.NET Performance. <http://msdn.microsoft.com/en-us/library/ff647768.aspx>.
- Lazy Initialization. <http://msdn.microsoft.com/en-us/library/dd997286.aspx>.
- Microsoft F# Developer Center: <http://msdn.microsoft.com/en-us/fsharp/default.aspx>.

- NModel 软件: <http://nmodel.codeplex.com/>.
- Our Pattern Language for Parallel Programming Ver 2.0.
<http://parlab.eecs.berkeley.edu/wiki/patterns>.
- 并行计算: <http://msdn.microsoft.com/en-us/concurrency/default.aspx>.
- Visual Studio 2010 并行性能分析: <http://blogs.msdn.com/b/visualizeparallel/>.
- .NET Framework 4 并行编程示例: <http://code.msdn.microsoft.com/ParExtSamples>.
- Windows Azure Guidance: <http://wag.codeplex.com/>.
- Performance Tuning with the Concurrency Visualizer in Visual Studio 2010. <http://msdn.microsoft.com/en-us/magazine/ee336027.aspx>.
- Scenario 主页: <http://code.msdn.microsoft.com/scenario>.
- Standardquery Operators Overview. <http://msdn.microsoft.com/en-us/library/bb397896.aspx>.
- TaskScheduler Events. http://msdn.microsoft.com/en-us/library/system.threading.tasks.taskscheduler_events.aspx.
- Unity Application Block: <http://msdn.microsoft.com/en-us/library/dd203101.aspx>.
- Windows HPC Server 2008: <http://www.microsoft.com/hpc/en/us/default.aspx>.

设计模式——.NET并行编程

本书简介

CPU计量器体现了一些问题。例如，某个内核的使用率为100%，而其他内核都是空闲的。或者你的应用程序是计算密集型的(即要占用大量CPU资源)，但你只使用了多核系统的一部分计算能力。如何解决这些问题呢？

简而言之，答案就是并行编程。像所有程序员一样，你可能熟谙编写顺序代码之道，但你会发现现在它不再满足你的性能要求了。要想有效地使用系统的CPU资源，必须把应用程序分割成块，这样应用程序就可以在同一时间运行。

这说起来容易做起来难。并行编程被誉为专家领域，它难以重现软件缺陷。

每个程序员似乎都有一些关于并行编程的趣事，可能由于某个神秘的错误，程序并没有像预期那样运行。在你编写并行程序时，这些故事应该能帮助你正视面临的问题和困难。幸运的是，微软给大家带来了帮助。.NET Framework 4引入了一种新的编程模型，大大简化了并行工作。其后台是复杂算法的支持库，在多核架构中动态分配计算。此外，Visual Studio 2010开发系统还包含调试和分析工具，以支持新的并行编程模型。而另一个帮助来源则是成熟的设计模式。本书借助任务并行库(TPL)和并行语言集成查询(PLINQ)，介绍了最重要和最常用的并行编程模型，并给出了这些模型的可执行代码示例。

作者简介

Colin Campbell是*Model-Based Software Testing and Analysis in C#*的合著者之一，他发表过数篇有关软件分析的论文。他是西雅图的Modeled Computation LLC的创始人和负责人。

Ralph Johnson是伊利诺伊大学的研究副教授。他是*Design Patterns*的四个合著者之一，并且是开发了第一个自动重构工具Smalltalk Refactoring Browser的项目组的组长。近几年来，他一直致力于记载并行编程的模式。

Ade Miller是微软的patterns & practices组的主力开发，他在这里管理了数个敏捷团队，这些团队为微软的客户提供项目上的实用指导。他的首要兴趣在于并行计算和敏捷软件开发实践。

Stephen Toub在微软的并行计算平台团队中工作。他致力于.NET和Visual Studio设计和开发下一代并发和并行编程模型。他所在的团队的博客地址如下：<http://blogs.msdn.com/pfxteam>。

清华大学出版社数字出版网站

WQBook 书文局泉
www.wqbook.com

ISBN 978-7-302-27997-6



定价：39.00元